Desarrollo de una Herramienta de Creación de Videojuegos de Rol Táctico para Escenarios Isométricos Compuestos por Bloques

Javier Druet Honrubia
Luis Alfonso González de la Calzada

GRADO EN INGENIERÍA INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO
EN INGENIERÍA INFORMÁTICA

Madrid, 13 de Septiembre de 2017

Director: Federico Peinado Gil
Codirector: Víctor Manuel Pérez Colado

Autorización de difusión y utilización

Javier Druet Honrubia y Luis Alonso González de la Calzada autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y el prototipo desarrollado.

Fdo. Javier Druet Honrubia

Fdo. Luis Alfonso González de la Calzada

Fdo. Federico Peinado Gil

Fdo. Víctor Manuel Pérez Colado

Agradecimientos

En primer lugar queremos agradecer a nuestro Director de Proyecto Fede por estar aguantando todas nuestras dudas, problemas y quebraderos de cabeza durante todo el curso.

Agradecer también a Victorma todas las horas que ha dedicado a cada apartado del proyecto, analizando las decisiones que tomábamos, guiándonos hacia el camino correcto y sobre todo enseñándonos a trabajar con su herramienta IsoUnity y con Unity. Sin él este proyecto hubiese sido mucho más complicado de llevar a cabo.

A nuestra familia y compañeros, por soportarnos explicando una y otra vez cómo funciona cada sección del proyecto, animándonos cuando nos atascamos y apoyándonos aunque no entendiesen una sola línea de código.

Por último agradecer a la Universidad Complutense de Madrid la oportunidad de llevar a la realidad este proyecto, por ofrecer los recursos y las personas necesarias para hacer de una idea muy básica un gran trabajo.

Índice

Contribución 109

Índice de figuras

Resumen

Gracias al avance de la tecnología y a una mayor disponibilidad de las herramientas, en estos últimos años hemos visto aparecer un gran número de proyectos de videojuegos desarrollados por estudios independientes. Algunos géneros tradicionales, como los juegos de plataformas o los de disparos en primera persona, cuentan con entornos de desarrollo específicos y muy potentes, que permiten prototipar ideas en cuestión de horas. Sin embargo, otros géneros de mayor variabilidad, como el de los juegos de rol, no cuentan con herramientas tan maduras y estables dentro de los motores de videojuegos de última generación.

Con el objetivo de ayudar a estos creadores independientes que desean desarrollar videojuegos de este género, se ha extendido una herramienta preexistente llamada IsoUnity, pensada para crear escenarios isométricos compuestos por bloques. Este proyecto amplía su funcionalidad, tanto a nivel de código como de herramientas de edición, para que se puedan crear con facilidad juegos de rol táctico con movimiento, combate por turnos y vista en perspectiva isométrica, al estilo de clásicos como *Final Fantasy Tactics*.

Con el objetivo de ofrecer a los usuarios una herramienta actual y versátil, se han estudiado y analizado un conjunto representativo de juegos del sector y en base a ese estudio se ha desarrollado el proyecto tratando de incorporar la mayoría de elementos comunes a todos, siguiendo una metodología clásica de análisis, diseño, implementación y prueba que ha dado lugar a un prototipo plenamente funcional que ilustra las posibilidades de esta herramienta. Actualmente el prototipo está en su primera versión (v 1.0), pero se estudia y ofrece la continua mejora de éste.

Este prototipo en su primera versión oficial ofrece las características más destacadas del conjunto de juegos estudiados haciendo hincapié en el sistema de turnos, la creación de personajes y habilidades y el movimiento de personajes basado en celdas desarrollado en IsoUnity.

Abstract

Thanks to the advances in technology and greater availability of tools, in recent years we have seen a large number of independent video game projects. Some traditional genres, such as platform games or first-person shooters, have specific and very powerful development environments that allow prototyping ideas in a matter of hours. However, other genres of greater variability, such as that of RPGs, do not have such mature and stable tools.

With the aim of helping these independent creators who want to create videogames of this genre, a preexisting tool called IsoUnity has been extended. This tool is designed to create isometric scenarios composed by blocks. The functionality of this tool has been expanded, both at the code level and editing tools, so that a designer can easily create tactical role-playing games with movement and combat by squares and isometric perspective view, in the style of classics like *Final Fantasy Tactics*.

With the aim of offering users a current and versatile tool, a representative set of games has been studied and analyzed, and based on that study the project has been developed, following a classic methodology of analysis, design, implementation and testing which has given rise to a fully functional prototype that illustrates the possibilities of this tool. At this time, the protoype is in its first version (v 1.0) but it is also work in progress.

The first version offers the most important features of the studied games: turn based gameplay, creating and editing characters and their abilities, and movement through cells in the map based on IsoUnity.

Capítulo 1: Introducción

Los programas y aplicaciones software de carácter lúdico llevan existiendo desde mediados del siglo XX. A lo largo de las últimas décadas las tecnologías utilizadas para desarrollar estos programas se han ido actualizando y mejorando, y con ello, las posibilidades de crear ideas totalmente innovadoras en el sector del entretenimiento han ido en aumento.

A medida que las tecnologías evolucionaban, los desarrolladores implementaron nuevos juegos cada vez más variados y complejos, pero al mismo tiempo llevar a la realidad todas estas ideas suponían un aumento enorme en el coste de desarrollo del juego.

A día de hoy desarrollar un videojuego comercial para el mercado de masas es arriesgado y conlleva muchos gastos, de manera que pocas empresas disponen del capital y los recursos como para realizar una producción a gran escala. En parte es debido a esto que han aparecido pequeños desarrolladores que buscan un tipo diferente de producción, algo más personal y también asequible en coste. A estos desarrolladores se les denomina “indies” (de “independientes”) y aunque muchas de sus obras no sean de la misma calidad en términos gráficos o contenido más extenso, estos proyectos destacan por el talento, la innovación y la dedicación con el que se crean. Es en este tipo de proyecto donde se quiere centrar nuestra aportación.

1.1 - Evolución de las herramientas de creación de Videojuegos

Si contemplamos la evolución sufrida por las herramientas de creación de videojuegos, veremos que en los primeros años de la Industria se dieron a conocer varias empresas que lograrían la fama gracias a proyectos muy innovadores, que daban a los usuarios nuevas e interesantes opciones a la hora de jugar delante de una consola. Atari comenzó siendo la empresa más famosa en el sector gracias al desarrollo de “Pong”, pero pronto la industria se diversificaría con la aparición de nuevos tipos de juegos (como las aventuras gráficas o conversacionales). Durante

los 80 aparecieron títulos como “Space Invaders”, que generaron importantes beneficios a las empresas productoras.

La industria japonesa se dio a conocer por todo el mundo gracias a “Super Mario Bros” [1] qué es el padre del género de plataformas. Además, aprovechando que en occidente acababan de aparecer los microordenadores, Nintendo presentó su consola NES en todo el mundo, durante la misma década de los 80. A partir de ese momento Japón pasaría a tener una relevancia global en el ámbito de los videojuegos.

Con la llegada de la década de los 90 la industria de los videojuegos sufrió una fuerte evolución en términos de complejidad multimedia. Durante los inicios de la década se comercializaron las primeras consolas portátiles. Game Boy ya existía, pero durante estos años experimentó su mayor auge.

Y es aquí donde comienza a tomar forma este proyecto, a medida que se mejoraron todas las herramientas para el desarrollo de videojuegos, se dejaron de lado las ideas de ofrecer a los desarrolladores utilidades para continuar con el género RPG y aunque aparecieron nuevos proyectos que apoyaban el desarrollo de estos videojuegos, no se fomentó su desarrollo en los motores de la época.

Cerca ya del siglo XXI se dió a conocer mundialmente la PlayStation y es a partir del año 2000 cuando el sector del videojuego da un salto enorme y comienza una mayor popularización entre jugadores de todas las características, edades y regiones. Los motores gráficos, aunque originarios de los 80, ahora se completaban con multitud de herramientas que permitían programar nuevos videojuegos de una manera más mucho más sencilla y cómoda, aunque la tecnología que hubiera por debajo fuese sumamente compleja. Por poner un ejemplo, Unreal Engine [2] permite trabajar con gráficos de mucha calidad de una manera relativamente sencilla hoy día, sin que el desarrollador tenga que preocuparse más de lo debido del rendimiento gráfico.

Cuando ya en el siglo XXI estos motores empiezan a popularizarse y a volverse trivialmente accesibles, cualquier persona con conocimientos bastante

básicos de desarrollo puede comenzar a probar sus ideas gracias a la potencia y flexibilidad de todas estas nuevas herramientas.

Es en este movimiento cuando surgió Unity [3] (2005), que, aunque ya se abordará en el estado de la cuestión, como muchos otros motores ofrece una cantidad enorme de herramientas al desarrollador independiente. De hecho, presume de ser uno de los entornos más multiplataforma, ya que ahora con la aparición de los nuevos *smartphones* la industria del videojuego ha seguido aumentando y haciéndose global.

1.2 - Propósito del trabajo

El propósito principal de este proyecto es doble. Por un lado se trata de crear una herramienta basada en Unity para el desarrollo de videojuegos de rol táctico, con movimiento y combate por turnos y estructurado en casillas, partiendo del trabajo realizado previamente con la herramienta IsoUnity [4]. En segundo lugar, consideramos imprescindible crear una demostración jugable con la propia herramienta desarrollada para poder ilustrar la utilidad y el funcionamiento de este proyecto.

Concretamente se implementarán nuevas funcionalidades que permitirán establecer las diferentes clases, propiedades y atributos que definirán a los personajes del juego y sus relaciones. Estas estructuras alimentarán los sistemas de combate y el desarrollo de la acción en las partidas, dando la posibilidad de crear en cada juego un planteamiento narrativo y estratégico distinto, incluso pudiendo ser extensible para innovar en lo que al género táctico se refiere. Entre estas funcionalidades está un sistema de gestión de objetos y armas, y un sistema de puntos para las diversas habilidades de los personajes, especialmente para las relacionadas con el combate.

Aunque la extensibilidad y la flexibilidad estarán siempre en el diseño del motor rol-táctico, con el objetivo de hacer una herramienta accesible, y siguiendo la filosofía en la que se desarrolló IsoUnity, el foco principal se centra en la realización de una capa de edición y autoría simple y efectiva. A través de dicha capa los desarrolladores no requerirán de conocimientos de programación para definir todos y cada uno de los elementos comunes a todo RPG táctico. Aquellos usuarios

expertos y desarrolladores, además de estas facilidades, siempre tendrán la puerta abierta a modificar las diferentes estructuras y flujos para que el motor se adapte mejor a sus necesidades.

La decisión de utilizar IsoUnity se fundamenta por un lado en que, el género del RPG táctico, como se analizará en el estado de la cuestión, suele estar muy arraigado a la perspectiva isométrica. Sin embargo, dado que el género incluye otros tipos de vistas, el desarrollo se orientará a ser una pieza independiente que podrá acoplarse a otros escenarios de juego a través de la implementación de diferentes adaptadores e interfaces. El adaptador que se acoplará a IsoUnity será el primer ejemplo de este hecho. Por otro lado, además, se pretende fomentar el desarrollo de videojuegos en perspectiva isométrica, que, aunque se ha visto recientemente impulsado por títulos como Monument Valley [5] o Chroma Squad [6] no está tan activo como en la edad de oro del software.

A través de este proyecto queremos ofrecer a toda la comunidad de desarrolladores independientes una herramienta completa pero sencilla que permita volver desarrollar juegos de rol con esta perspectiva. Títulos tan conocidos como *Final Fantasy Tactics* [7] o *Fire Emblem* [8] son muy queridos por los jugadores más veteranos y esperamos que nuestro trabajo sirva para que volvamos a ver lanzamientos de obras tan estimulantes como estas.

Capítulo 2: Revisión del estado de la cuestión

Para revisar el estado de la cuestión que aborda este proyecto, analizaremos los siguientes puntos base, explorando los diferentes ejemplos y trabajos que se han realizado al perfecto y destacando la importancia de cada uno de ellos:

1. Videojuegos del género rol tácticos e isométricos, abarcando los títulos destacados históricamente, aquellos más innovadores y los títulos más recientes.
2. Entornos de desarrollo, motores y herramientas que facilitan este tipo de desarrollo.
3. Estudio específico de *IsoUnity*, herramienta base sobre la que asentará este proyecto.

El orden de los puntos se corresponde al análisis realizado para el asentamiento inicial y descripción de los requisitos del proyecto. Para ello, se analizan los puntos clave de los juegos más importantes de este género, para después centrarnos en las herramientas que otorguen más facilidad y flexibilidad para desarrollar el trabajo. Finalmente se buscarán complementos que ayuden a perfeccionar el proyecto.

2.1 - Videojuegos referentes en el género

Con la idea de desarrollar un videojuego de rol con movimiento y combate en perspectiva isométrica se ha tomado la decisión de estudiar cada uno de los títulos más importantes de este género y de otros géneros cercanos. La metodología del estudio se centra por un lado en destacar los puntos a favor y en contra de generalizar lo visto en cada videojuego al motor, detallando su mecánica de juego y, por otro lado, en analizar la manera de mejorar o modificar los errores o carencias que hayamos podido encontrar en estos. Por ejemplo, se ha intentado analizar una serie de juegos muy exitosa, *Fire Emblem*, que aunque no utiliza perspectiva isométrica, tiene muchas ideas innovadoras en el género de rol táctico por turnos, y por ello se le dedicará una subsección en este capítulo.

2.1.1 - Final Fantasy Tactics

La saga de juegos Final Fantasy Tactics es uno de los mayores referentes que tiene este proyecto. Esta saga de juegos (Final Fantasy Tactics (1997), Final Fantasy Tactics Advance (2003), The War of The Lions (2007), Advance 2: Grimoire of the Rift (2008)) [9] tiene gran parte de la funcionalidad y de los aspectos técnicos que se quieren implementar durante el desarrollo del trabajo.

Esta saga fue creada en un principio por SquareSoft en 1997 y los demás títulos fueron desarrollados por el equipo Square-Enix que surgió de una fusión de dos empresas, Square y Enix [10]. Aunque comenzó como un videojuego en solitario, la empresa decidió unir todos los juegos mencionados en una recopilación llamada *Ivalice Alliance* [11] Esta saga de videojuegos se define por pertenecer al género RPG táctico, con una ambientación de “espada y brujería” muy fantástica, y dispone de una serie de características que nos parecen básicas para este tipo de software.

De esta saga de videojuegos se han de destacar dos temas muy interesantes que se explican con sumo detalle:

1. **Combate**:
   1.1. **Turno**: el sistema de combate se basa en una división por turnos según las estadísticas de los personajes de cada bando, en la que cada combatiente tiene una serie de habilidades especiales que pueden usar una vez por turno, siempre que sus puntos de vida y de maná (magia) se lo permitan. En cada turno, cada bando va posicionando sus personajes a medida que les llega el momento y se van enfrentando entre sí hasta que el ganador cumple el objetivo del combate. El objetivo del combate puede ser muy variado: eliminar al jefe de los rivales, eliminar a todos los enemigos, cumplir una serie de turnos, etc.

   1.2. **Posicionamiento**: El posicionamiento del personaje causa efectos en la acción a realizar: si el jugador coloca a uno de sus combatientes detrás de sus enemigos, éste tendrá más probabilidad de golpear al rival, además recibirá una bonificación si ataca al enemigo con armas

que ataquen a la debilidad del contrario (un guerrero recibe más daño si se le ataca con magia, por ejemplo). En el juego también existen personajes que pueden atacar a distancia y a los que el posicionamiento no les afecta tanto, ya que pueden evitar el combate cuerpo a cuerpo, por lo que se valora que el jugador disponga de un ejército con varios tipos de luchadores y que planifique las batallas antes de iniciarlas.

1.3. **Experiencia**: También a medida que se va avanzando el juego, los personajes del jugador van adquiriendo experiencia que permite mejorarlos y modificarlos. De esta manera cada personaje puede cambiar de oficio y obtener nuevas habilidades (por ejemplo, un mago se puede convertir en hechicero, lo que le confiere habilidades más estratégicas y potentes).

1.4. **Equipamiento**: Por último, y como parte esencial del sistema de combate, al jugar se van consiguiendo nuevas armas y objetos que ayudan a mejorar a los combatientes. Las armas con que se equipan a los luchadores hacen que éstos sean más fuertes a través de la modificación de sus estadísticas. Además, a través del uso de cada arma los personajes pueden desbloquear nuevas habilidades (por ejemplo, el uso de un cetro de fuego confiere al jugador el hechizo bola de fuego).

1.5. **Reglas de combate**: Además se agregó a esta saga de videojuegos una serie de normas de combate según la zona, el momento de la historia y los personajes que pertenecieran a los diferentes bandos, lo que permite dar al sistema un nivel de complejidad mayor para los jugadores más avanzados.

2. **Historia:** La historia se va completando a medida que el jugador completa diferentes misiones, que podía obtener en las tabernas de las ciudades que existen a lo largo de todo el mapa del juego. Este mapa también se utiliza para completar diferentes misiones opcionales que sirven para conseguir nuevos seguidores en tu ejército, completar objetivos de la

historia para obtener información secreta o conocer más acerca del mundo del juego. Nos ha interesado esta idea ya que el jugador tiene una cierta libertad para disfrutar del juego y no seguir una historia lineal que pueda dejarle sin opciones a la hora de saber más acerca de la historia del juego o indagar detalles ocultos a lo largo del mapa. Esta característica es muy importante para el proyecto ya que disponemos de una herramienta de diálogos y narrativa de *IsoUnity* que permite al usuario la total libertad de narrar una historia única en su videojuego.

2.1.1.1 Mapa Overworld

2.1.2 - Saga Diablo

Diablo es uno de los videojuegos de rol más más famoso que representa a la perfección nuestra idea de perspectiva isométrica. Se trata de toda una serie de juegos pertenecientes al género de estudio que ha sido desarrollada por Activision Blizzard [12]. A la hora de profundizar en los detalles de esta saga, nos vamos a centrar en un aspecto: el sistema de inventario.

Diablo dispone de uno de los sistemas de inventario que más se utilizaban antiguamente, y que siguen funcionando bien a día de hoy. Básicamente, el personaje del jugador no tiene un “peso” que limita la cantidad de objetos que el jugador puede cargar (como en Skyrim [13]), si no que dispone de un espacio en el que guardar todo el botín, si este no se puede guardar dentro del espacio de la mochila, no podrá llevarlo. Es un sistema muy interesante ya que obliga a gestionar muy bien qué llevar y qué no llevar a cada misión, por lo que cada jugador

desarrolla una manera particular de prepararse para las misiones. Además, así se puede limitar el avance que un jugador puede conseguir al repetir una misión varias veces, ya que sólo podrá extraer de la misión un número determinado de objetos que le serán útiles para su personaje.

2.1.2.1 Inventario de Diablo II

En el sistema de Diablo, los objetos ocupan una serie de celdas o ranuras en la rejilla del inventario (por ejemplo, una espada puede ocupar 2x4 celdas, dos de ancho y 4 de alto) por lo que guardar los objetos correctamente es también necesario para optimizar el desarrollo del personaje y aprovechar bien su equipamiento. Además del inventario el jugador dispone de unos espacios extra para “vestir” a su personaje, lo que permite asignar unos objetos equipados (un casco, una armadura, una o dos espadas en cada mano…), que son los que utilizará durante las batallas. Con este sistema se consigue que el jugador enfoque a su personaje en la búsqueda del mejor equipamiento para sus habilidades, ya que muchas se potencian con los objetos correctos.

Además este sistema también disponía de una “moneda” del juego, con la que el jugador podía ir mejorando a los trabajadores de las ciudades para que te ofrecieran nuevas armaduras o nuevas mejoras para ayudarte a avanzar en la historia. La moneda del juego puede tomarse como referencia para implementar un sistema de comercio que permita al jugador obtener objetos que le ayuden a lo largo de su aventura. Además, el sistema de moneda permite dar valor a todo lo conseguido por el jugador y le permite saber “de un vistazo rápido” cómo de bueno es el equipamiento o los objetos que ha conseguido en sus hazañas.

2.1.3 - Path of Exile

Aunque muchos consideran que es idéntico al Diablo, Path of Exile [14], utilizando un mismo sistema de combate, mismo sistema de inventario y mismo sistema de clases, consigue marcar una diferencia muy grande mediante el uso del enorme árbol de desarrollo del que dispone cada personaje en el juego.

Este árbol de desarrollo permite a cada jugador definir las habilidades, características y, en resumen, el futuro del personaje de una manera nunca vista en este estilo de juegos: la libertad con la que se puede exprimir cada detalle del árbol, experimentar y probar cada una de las ramas, siendo además todas completamente diferentes. Los jugadores de *Path of Exile* tienen la posibilidad de centrar su personaje en un objetivo que le permita utilizar las habilidades de una manera única. Es nuestra intención realizar un esquema similar al desarrollado en este juego, pero con las habilidades. Mediante los requisitos que ya explicaremos más adelante, podremos crear una estructura enorme de desarrollo de habilidades en las que a medida que el personaje progresa adquiriendo experiencia y equipamiento, desbloquea nuevas habilidades, ofreciendo así un desarrollo y un sistema de progresión a lo largo de la historia. Por ejemplo, un personaje que sea nivel 10 podría desbloquear una nueva habilidad pasiva que le permita equipar dos armas, o lanzar una habilidad de daño en área.

2.1.4 - Bastion

Bastion [15], al igual que los juegos anteriormente comentados, vuelve a hacer uso de la perspectiva isométrica para mostrar un mundo dividido en niveles en

los que el jugador debe avanzar por ellos para completar los desafíos. Este juego mezcla dos géneros, tanto el RPG como las plataformas, ya que, aunque tenga bastantes parecidos con otros títulos, tiene un estilo único que le ha otorgado una gran fama y respeto en la comunidad.

De este videojuego se quieren tomar como referencia las ideas que utilizó el equipo de SuperGiant Games [16] para convertirlo en multiplataforma, ya que realizaron un gran trabajo de diseño a la hora de permitir que consolas, PC y móviles pudiesen disfrutar de este gran juego.

2.1.5 - Fire Emblem

Esta saga de juegos se caracteriza por la necesidad de realizar las batallas de la manera más estratégica posible, ya que se suelen desarrollar combates con una gran cantidad de participantes. El punto más importante que queremos destacar para el proyecto es la relación que existe entre los personajes que el jugador controla. Todos ellos tienen relaciones sociales y emocionales con los demás miembros del grupo, con lo que se pueden llegar a tener amigos que, cuando están a cierta distancia en un combate, pueden protegerse o realizar ataques especiales combinados.

2.1.5.1 Batalla en Fire Emblem

Por otro lado este sistema que desarrolla relaciones entre personajes da lugar a nuevos héroes que se pueden controlar. En este juego un matrimonio puede llegar a tener un hijo, que podrá ser jugable a lo largo de la historia cuando haya crecido.

2.1.6 - *League of Legends, Heroes of the Storm y DOTA 2*

Aunque no se suele asociar los juegos del tipo MOBA (Multiplayer Online Battle Arena) [17] a los RPG, hemos decidido incluirlos por la personalización a la que puede someterse el personaje del jugador a lo largo de cada partida. Según contra quien se enfrente el jugador, a medida que avance el combate, este irá personalizando sus habilidades y objetos para prepararse contra los enemigos.

Para que sea comprensible, en *League of Legends* o *DOTA 2,* a medida que avanza la partida se consigue oro, que a su vez permite comprar equipamiento para enfrentar a los enemigos utilizando diferentes estrategias. Un jugador puede mejorar su velocidad de ataque para realizar mucho daño en pocos segundos o por el contrario mejorar su armadura para recibir menos daño de los ataques básicos.

Por otro lado, en *Heroes of the Storm*, se planteó un nuevo sistema basado en “talentos” que permiten adaptar las habilidades de los personajes a las diferentes situaciones de combate. En ese aspecto hemos ideado el sistema de requisitos para que cada personaje disponga de unas habilidades diferentes según su clase y su especialización, nivel o equipamiento.

Esta es una característica principal que definitivamente queremos añadir a nuestro proyecto, ya que cuando se desarrolla un videojuego suele ir ligado a una historia. Queremos que esa capacidad de personalización y mejora de los personajes se pueda ver reflejada a medida que la historia avanza.

Por ello, se otorgará a los personajes de una serie de características y atributos que se podrán ir modificando según la equipación y habilidades del personaje para que estas mejoras tengan un significado en la historia y se pueda otorgar al jugador de un sentimiento de mejora y adaptación antes diferentes situaciones durante el desarrollo del juego.

2.1.7 - *Saga Little Big Adventure*

La saga de aventuras *Little Big Adventure* (1994-1997) es un buen representante en el uso de la perspectiva isométrica de varias alturas con tiling para crear un juego con una fuerte narrativa. Cuenta la historia de Twinsen, un habitante del planeta Twinsun, el cual el dictador FunFrock domina en su hemisferio norte con mano de hierro. Cuando Zoe (la novia de Twinsen) es raptada por FunFrock, Twinsen tendrá que enfrentarse al dictador y a su ejército de clones para devolver la paz a Twinsun.

Si bien se aleja del concepto de RPG tradicional (comparte más características con el género de plataformas) incluye características como inventario, habilidades y mejora de estadísticas del personaje que sí son propias de un RPG. La simplicidad con las que se usan estos elementos ejemplifica lo interesante que puede llegar a ser un juego RPG sin necesidad de ser muy complicado.

2.1.7.1 Screenshot de Little Big Adventure 2

2.1.8 - *The Walking Dead: Road to Survival*

The Walking Dead [18] nace como un cómic mensual que cuenta la historia de un grupo de supervivientes tras un apocalipsis zombi. En 2010 recibió una adaptación a la televisión convirtiéndose en un fenómeno de masas. El merchandising no tardó en hacer acto de presencia y el desarrollo de videojuegos también era cuestión de tiempo.

En 2015 se publicó *The Walking Dead: Road to Survival* [19] para dispositivos móviles. Este videojuego se focaliza en el combate táctico de mecanismos simples en batallas en las que se van consiguiendo personajes, armas e ítems usando de trasfondo la historia de los cómics.

1. **Combate.** El jugador maneja a un grupo de varios supervivientes en el que cada personaje puede atacar solamente cuerpo a cuerpo o a distancia. Tras varios ataques, un personaje puede utilizar su habilidad especial. Las batallas pueden ser contra humanos o contra zombis. Los primeros actúan de forma similar al grupo que portamos. Los zombis actúan en grupos mayores pero sólo atacan cuerpo a cuerpo y se van acercando poco a poco. El escenario puede dificultar ciertos ataques. El jugador debe pensar qué equipo de supervivientes, armas e ítems debe llevar a la batalla, cómo administrarlos en ésta y estudiar a los enemigos y al escenario para salir victorioso. Si bien la vista no es isométrica, el combate táctico es un buen ejemplo para el desarrollo de nuestra herramienta.

2.1.8.1 Batalla vs zombis

2. **Personajes, armas e ítems.** Al ganar una batalla el jugador consigue varias recompensas: “rescatas” personajes y obtienes armas e ítems. Todos estos elementos se pueden intercambiar en el mercado o mezclarlos para conseguir personajes e ítems más poderosos. Gracias a esto se pueden superar batallas más difíciles y así seguir avanzando en la historia.

2.1.8.2 Configuración del grupo o *party* (personajes, armas e ítems)

2.1.9 - *Fallout, Fallout 2 y Fallout Tactics*

La saga de juegos Fallout [20] es uno de los mayores representantes de los RPG tradicionales desde que en 1997 apareciese el primer *Fallout* hasta que en 2015 llegó *Fallout 4*. Ambientados en una posguerra nuclear durante los siglos XXII y XXIII, sus dos primeros títulos se representan en perspectiva isométrica y ataque por turnos antes de que los siguientes diesen el salto a las 3D y a otro sistema diferente para los combates. También hay un juego entre estas dos épocas llamado *Fallout Tactics* que desarrolla un sistema de combate por turnos más completo.

De esta saga nos focalizamos en los tres primeros títulos, concretamente en el sistema de diálogos y en los atributos de los personajes. Además estudiamos las mecánicas de combate:

1. **Sistema de diálogos.** Siguen el esquema de una aventura gráfica clásica en el sentido de que permiten varias posibilidades para responder, guiando así la conversación por distintos derroteros según tus intereses. Además la representación de estos diálogos muestra claramente los actores implicados.

2.1.9.1 Diálogos en Fallout 2

2. **Atributos de los personajes.** Fallout permite configurar los atributos del personaje principal. Estos atributos definen su personalidad y cualidades físicas, limitando o potenciando ciertas acciones en el mundo del juego.

2.1.9.2 Configuración de los atributos del personaje en Fallout 2

3. **Mecánicas de combate por turnos:** *Fallout Tactics* introduce dos nuevos sistemas de combate al modo *Individual Turn-Based (ITB)* de los Fallout originales: *Continuous Turn-Based (CTB)* y *Squad Turn-Based (STB)*. Si en ITB los personajes actúan por turnos según un atributo llamado *Secuencia*, en CTB, cualquiera puede actuar a la vez y las acciones se pueden repetir según el atributo de *Agilidad* del personaje en cuestión. STB es una variación de ITB, pero cada turno se asigna a un equipo. Otros cambios que no tienen que ver con turnos pero que afectan a la táctica son la posibilidad de estar tumbado, agachado, o de pie, y de usar personajes a modo de centinelas para cubrir al resto del equipo mientras estos hacen otras cosas.

2.1.10 - *Zelda II: The Adventure of Link*

La saga de Zelda es una de las más conocidas de Nintendo y se extiende en el tiempo desde 1986 a la actualidad. Aunque por sus características no es solamente un RPG al uso, sino que lo combina con acción, sí que tiene un gran componente de este género.

Esta revisión se centra exclusivamente en *Zelda II: The adventure of Link* publicado en 1988. Aunque es considerado la “oveja negra” dentro de la saga Zelda, introdujo una gran cantidad de novedades. Es interesante la forma en la que moverse por ***un gran mapa*** guía el desarrollo del juego conectando distintas áreas, y ***cómo se pasa desde esta perspectiva a otra*** lateral para las batallas o la exploración de poblados. De este concepto se pueden tomar ideas para desarrollar la narrativa del juego diferenciando las batallas tácticas a nivel isométrico que desarrollaremos en la herramienta de una exploración en un mapa también isométrico pero libre de turnos, explotando así las capacidades que ofrece IsoUnity.

1. **El mapa**: la perspectiva top-down (representación desde arriba de una cuadrícula clásica) muestra el gran mundo que Link debe explorar. Si Link se mueve por los caminos marcados no encontrará enemigos, pero si se adentra en bosques, desiertos u otros lugares unos personajes oscuros aparecerán para asaltarle y si le alcanzan Link deberá pelear para seguir adelante.

2.1.10.1 Mapa del mundo en Zelda II

2. **El cambio de perspectiva**: si Link es alcanzado por esos enemigos oscuros mientras se mueve por el mapa entonces la perspectiva cambia y se entra en modo batalla. Dependiendo de la zona del mapa en la que Link sea alcanzado por los personajes oscuros (bosque, desierto, pradera...) y de qué personaje le alcanza (uno pequeño o uno grande), la batalla será distinta en escenario y enemigos.

2.1.10.2 Link perseguido en el mapa del mundo vs Link en batalla

2.1.11 - *Baldur's Gate*

Basado en las reglas de *Dungeons & Dragons* [21], juego de rol clásico por antonomasia, *Baldur's Gate* [22] sigue las andanzas del personaje principal en un mundo fantástico en un intento por descubrir el porqué de las extrañas cosas que están sucediendo.

Lo interesante de este juego es que adapta el sistema de un juego de rol de mesa como *Dungeons & Dragons* para en el que el devenir de los personajes viene determinado por su ***alineamiento, habilidades, reputación y reclutamiento*** entre ellos. Los RPG tácticos, a excepción del uso de habilidades, normalmente no juegan con estos conceptos que representan mejor el carácter social del personaje y su posible fidelidad a lo largo del combate, siendo este un buen apartado en el que innovar dentro del proyecto, pudiendo plasmarlo en los personajes y sus actitudes de forma simplificada.

1. **El alineamiento:** una forma de clasificar a los personajes según si respetan la ley (legal, neutral o caótico) y si tienen una buena moral (bueno, neutral o malo). Con estos datos se pueden crear narrativas y potenciar (o minimizar) los atributos del personaje y su evolución.

2.1.11.1 Ejemplos de alineamiento [23]

2. **Las habilidades:** para definir un personaje nos basamos en unas habilidades centrales, o habilidades *core*. Aquí son *fuerza, destreza, constitución, inteligencia, sabiduría* y *carisma*, pero se podrían definir otras.
3. **La reputación:** representa el alineamiento global de un grupo de personajes que trabajan juntos (el *grupo* o *party*). Un personaje con un alineamiento muy alejado del general del grupo puede causar conflictos e incluso llegar a abandonarlo.

4. **Reclutamiento:** la historia avanza de manera distinta dependiendo qué personajes se recluten y qué personajes no.

Estos apartados pueden implementarse dentro de la historia o como mecánica de juego. Por ejemplo: si bien un personaje puede ser amistoso si se le paga bien, si el personaje no recibe buenos objetos o buenas pagas y se abusa de él en los combates, éste podría ser receloso a arriesgar su vida por su capitán.

2.1.12 - *Chroma Squad*

En *Chroma Squad* contamos con un grupo de actores que interpretan unos personajes muy parecidos a los famosos Power Rangers. Es una mezcla de juego de gestión para hacer que el estudio que produce las series prospere, con un juego de rol táctico en el que las batallas por turnos son las grabaciones de los capítulos. Su reciente publicación (2015, Behold Studios [24]) y su buena acogida son un buen ejemplo para entender cómo un producto de RPG táctico puede funcionar en el mercado actual.

2.1.12.1 Una batalla de Chroma Squad

Es evidente que un juego de este tipo funciona por su ***originalidad*** a la hora de adaptar la historia a un RPG táctico. Por ejemplo, la posibilidad de acabar una batalla de una manera más difícil viene motivada por conseguir más fans e ingresos. El jugador no es ningún héroe, sino el director de los capítulos. De esto podemos concluir que para desarrollar una herramienta que permita hacer RPGs tácticos hay

que tener en cuenta que debe ser muy versátil, permitiendo crear cosas como *Chroma Squad* sin caer en nomenclaturas propias del rol fantástico, de la jugabilidad y la ambientación más predecibles.

Otro punto a destacar es que el juego está desarrollado en Unity, motor sobre el que está desarrollado IsoUnity y sobre el que desarrollaremos nuestra herramienta.

2.1.13 - *Wafku / Dofus*

Conocido por ser uno de los pocos RPG tácticos MMORPG (massive multiplayer online RPG), *Wafku y Dofus* [25] se basan en el desarrollo de cuatro naciones (potenciadas por los jugadores) en las que se debe ir mejorando al personaje a medida que va subiendo de nivel mientras el jugador se mezcla con la comunidad para trabajar en equipo y superar las mazmorras.

Queremos destacar de este videojuego su ***carácter humorístico y su estilo multijugador***, pues no conocemos muchos juegos que permitan unirse a otras personas para conseguir los objetivos del juego. La funcionalidad multijugador permite analizar los sistemas de turnos aplicados a un entorno de red, lo cual es interesante de cara al futuro de la herramienta desarrollada.

2.1.14 - *Sacred Fire*

Sacred Fire es un juego disponible en *Steam Direct* [26] que parte de la toma de ***decisiones en el combate***. Se ha tenido en cuenta este juego ya que es un punto de vista diferente a todo lo visto anteriormente. El sistema de ***combate es novedoso*** y a la vez muy interesante, con dinámicas innovadoras y en general un sistema complejo en el que el jugador debe tener en cuenta muchos factores para poder alcanzar la victoria.

El juego dispone de un sistema de progreso de personajes que permite ir mejorándolos a medida que se avanza en la historia, con la posibilidad de desbloquear nuevos compañeros. Para mejorar a los personajes se obtienen unos puntos que permiten personalizar la personalidad de los jugadores, como es un

RPG psicológico, estas personalizaciones tratan de preparar a tu personaje para situaciones que tienen que ver con las traiciones, lealtad y venganzas.

2.1.14.1 Mejora de personajes en Sacred Fire

Hemos querido destacar la toma de decisiones durante los combates ya que se desenvuelve de una manera única. El jugador debe elegir constantemente entre difíciles opciones para ir desarrollando la historia y las luchas. Por ejemplo, los combates no son una lucha típica, se basan en toma de decisiones. Se va ofreciendo al jugador una serie de opciones y según cual se haya seleccionado, el personaje puede recibir o provocar daño. En una pelea, se podría dar la opción de acercarse al enemigo, girar a la derecha o gritarle y según que opción se escoja, el combate se desenvuelve de una manera diferente.

2.1.15 - *The DragonLoft (Hellenica)*

The DragonLoft es un RPG Táctico que se basa su sistema de combate de manera similar a los juegos ya mencionados, pero con un detalle que se debe tener en cuenta: ***el sistema de combate***. Funciona por equipos en dos fases, el primer equipo planifica todos los movimientos de sus miembros, y cuando termina la fase de planificación, se realizan las acciones indicadas, después, el equipo contrario

planifica su fase de acciones y se llevan a cabo. Este es un sistema de turnos que tenemos muy en mente, ya que provoca que el jugador planifique todas las posibilidades que tiene el enemigo para realizar su turno. Eso significa que el carácter estratégico de este título sobresalga por el resto, ya que una posibilidad no calculada puede provocar la derrota del jugador.

Se quiere destacar este tipo de combate ya que requiere que el jugador planifique a la perfección todos sus movimientos y prepare las defensas para el turno del contrario. En este proyecto, queremos destacar que el planteamiento estratégico de cada turno es de vital importancia si se quiere tener éxito en la campaña, por ello queremos remarcar que aunque estamos constantemente hablando de RPGs, no nos podemos olvidar de que la estrategia y planificación son los que hacen del título un videojuego entretenido.

2.1.15.1 Screenshot del juego

Por otro lado, el juego dispone de un ***sistema narrativo*** bastante interesante, ya que ofrece conversaciones tanto en la lucha como fuera de ella y según como vayan aconteciendo los combates, podrán ocurrir unas tramas u otras. Se quiere centrar el proyecto en el sistema de combate, pero no se pueden olvidar otros aspectos importantes.

2.2 - Herramientas que ofrecen el desarrollo en perspectiva isométrica y/o RPG

En este capítulo revisamos las herramientas más populares y relevantes de cara a este trabajo, diseñadas para facilitar el desarrollo de videojuegos de rol, de títulos en perspectiva isométrica, o ambas cosas a la vez.

2.2.1 - Múltiples versiones de RPG Maker

Desarrollada por ASCII Corporation y Enterbrain [27], quizás sea la herramienta más importante a la hora de entender cómo debe funcionar un sistema que proporcione al diseñador la posibilidad de crear un juego de rol con mínimas necesidades de programación. *RPG Maker* ha tenido distintas versiones desde 1995 hasta la actualidad (***95, 2000, 2003, XP, VX, VX Ace,*** y ***MV***).

RPG 95/2000/2003: RPG 95 es el primero de la saga y RPG 2000 es una evolución que ofrece mejoras técnicas y visuales. RPG 2003 es una pequeña modificación de RPG 2000 que proporciona batallas en perspectiva lateral y sistema de combate ATB (Active Turn Battle, similar a los Final Fantasy de Super Nintendo).

RPG XP: primer RPG Maker que incluye propiedades de capacidad de programación. Para ello usa Ruby. Incluye evoluciones técnicas y visuales pero cambia la visualización del combate a una vista frontal basado en sprites estáticos con fondos de pantalla llamados battlebacks. Simplificado al extremo, es con las capacidades de programación donde se pueden conseguir distintos sistemas de batalla, ATB, CTB (Conditional Turn-Based Battle) o ADB (Active Dimension Battle, tipo Final Fantasy XII).

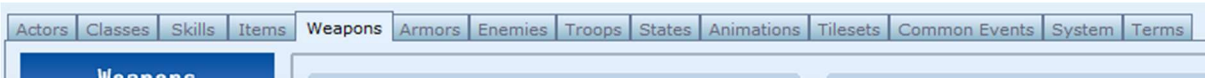
RPG Maker VX: consigue aunar lo mejor de RPG Maker 2003 (recupera las posibilidades gráficas de este) y RPG XP (sigue dando soporte a programadores con ligeras mejoras). Añade los sistemas de combate Optima o Command Sinergy (estilo Final Fantasy XIII).

RPG Maker VX Ace: evolución de VX añadiendo mayor personalización

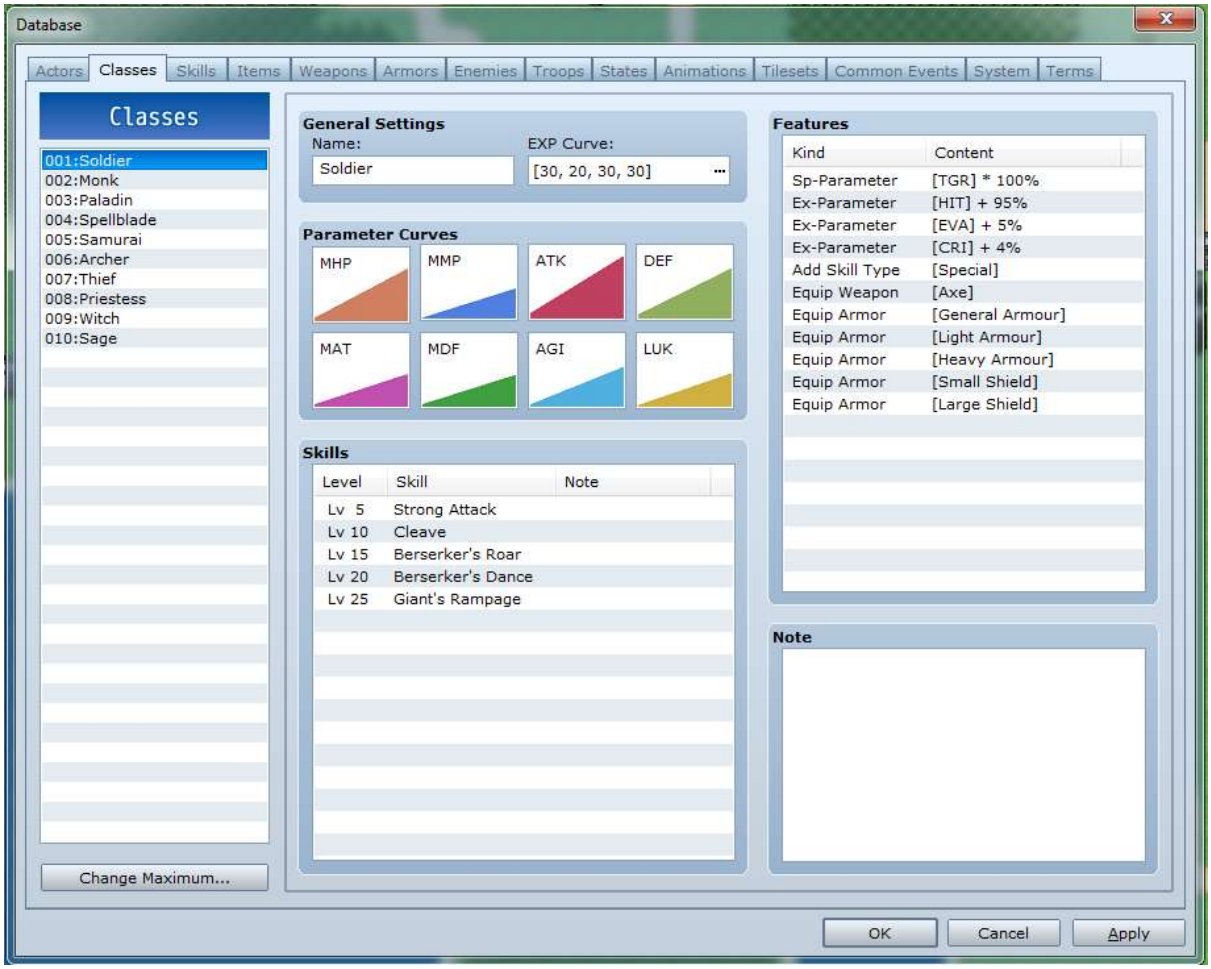
gráfica, cambio de interfaz y motor de acciones. Ataques, conjuros y objetos pueden tener sus propias fórmulas de daño y recuperación.

RPG Maker MV: constituye una de las mayores mejoras. Gráficos de alta resolución, edición multiplataforma (Windows y MAC), distintos sistemas de combate (ATB, CTB, ADB, y Optima) y visión de las batallas (frontal, lateral). Cambio del lenguaje de programación a Javascript. Los juegos creados pueden ser jugados en PC (Windows, Macintosh o Linux), aunque hay versiones específicas de RPG Maker para Nintendo 3DS, Game Boy, PlayStation y demás consolas.

Para dar soporte a todas las posibilidades descritas, RPG Maker posee un potente conjunto de editores que permiten personalizar hasta el más mínimo detalle del juego: personajes, enemigos, armas, batallas, escenarios, etc. Tomando estos editores como ejemplo podemos tener una visión clara de lo que una herramienta de creación de RPGs (y por extensión RPGs tácticos) debe tener (Figuras 2.2.1.1 y 2.2.1.2).



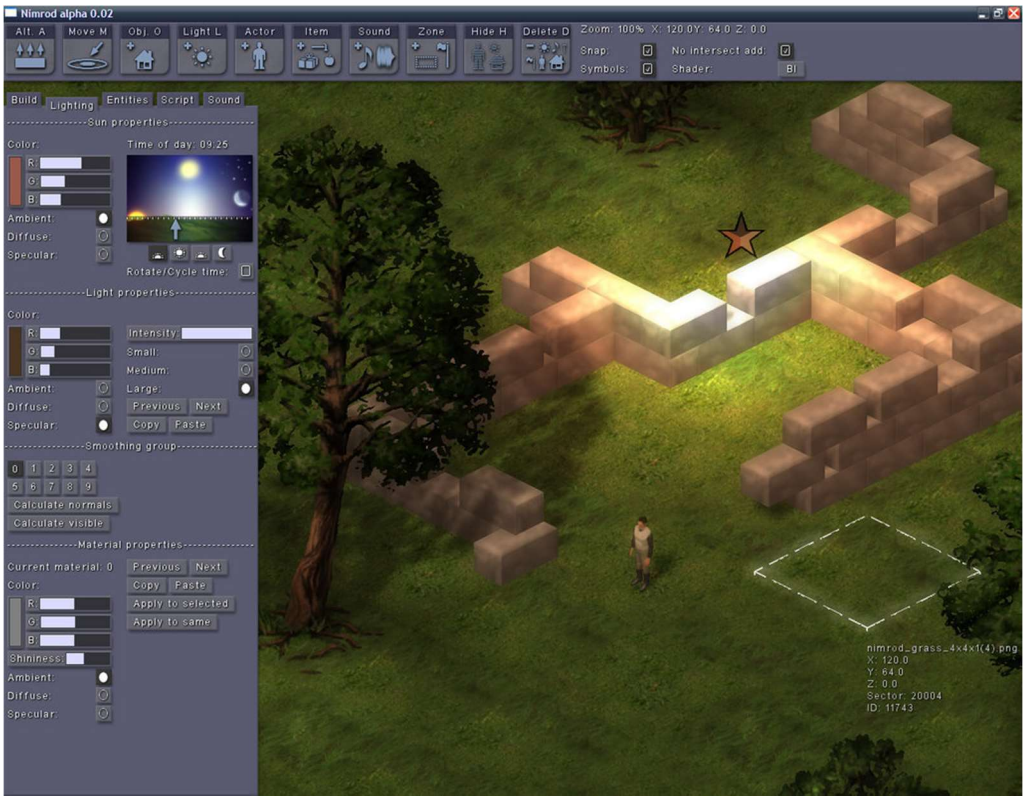
2.2.1.1 Detalle del dock de todos los editores de RPG Maker VX Ace



2.2.1.2 Pestaña “Classes” de RPG Maker VX Ace donde se aprecian fórmulas de crecimiento de parámetros, lo que ocurre al subir niveles de clase…

2.2.2 - *Nimrod*

Herramienta desarrollada por un programador independiente, que permite la creación de mapas en perspectiva isométrica con la capacidad de implementar la hora del día exacta en los mapas, de tal manera que el usuario puede establecer el paso del tiempo para que el efecto de las luces y sombras afecten a la escena.



2.2.2.1 Detalle del editor de Nimrod

Como muestra la figura, el editor está bastante completo para ser desarrollado por una sola persona, aunque como ya hemos mencionado, le otorga importancia sobre todo al factor de la iluminación en los mapas.

2.2.3 - *Wurfel Engine*

WurfelEngine [28] es un motor de desarrollo de videojuegos isométricos de código abierto implementado en ***Java***. Ha sido desarrollado por un estudiante de Informática llamado Benedikt S. Vogler (BSVogler en GitHub) y es apto para ***múltiples plataformas*** (Windows, MAC y Linux).

Usa algoritmos especialmente diseñados para funcionar sobre escenarios isométricos, renderizando sólo lo que ve el usuario y la iluminación puede ser vía vertex o vía normal map. Tiene un ***editor WYSIWYG*** (What You See Is What You

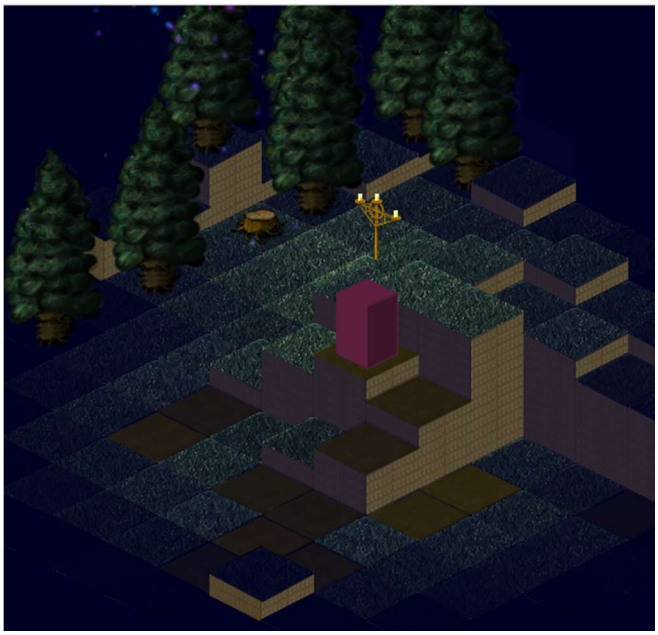
Get) que permite crear escenarios de forma sencilla. El motor no incluye físicas ni scripting de manera predeterminada.

2.2.5.1 Editor de Wurfel Engine

2.2.4 - *JSIso*

JSIso [29] es un motor bajo licencia MIT desarrollado por Iain Hamilton en ***Javascript***. Basado en tiles, usa el canvas de HTML5 para mostrar sus gráficos. Está en sus primeras etapas de desarrollo aunque éste avanza de manera constante y ya tiene implementadas una gran cantidad de funcionalidades.

Da la capacidad de construir fácilmente complejos mapas isométricos. Puede hacer uso de ***archivos de formato de Tiled Editor*** (programa externo) para crear los suyos. También tiene capacidad de auto-scaling, inclusión de spritesheets, zooming, sistema de colisiones simples, niebla (limitación de la visión), AI-pathfinding, y rotación de los mapas.



2.2.6.1 Mapa creado en JSIso

2.2.5 - *IsoHill*

IsoHill [30]es un motor isométrico desarrollado por Jonathan Dunlap en ***Actionscript para Flash Player 11*** sobre el framework Starling 2D.

Incluye soporte para plugins que modifiquen el motor, soporte para el conocido editor Tiled, lógica para los sprites isométricos, capas ilimitadas, antialiasing y mipmapping. Está pensado para crear robustos juegos de navegador.

2.2.7.1 Mapa creado en IsoHill

2.3 - *IsoUnity*

IsoUnity es una herramienta para *Unity* que permite al desarrollador crear escenarios en perspectiva isométrica de una manera sencilla y con un gran nivel de personalización. Gracias a sus editores y todas las opciones implementadas es realmente fácil crear un mapa en el que varios personajes se desenvuelven y realizan varias acciones (mantener una conversación, movimiento, colisiones…).

Se trata de una herramienta clave porque la intención de este proyecto es precisamente integrar la herramienta RPG que se va a desarrollar (de ahora en adelante *TRPGMaker*) con *IsoUnity*, con la finalidad de ofrecer a los desarrolladores una herramienta completa y genérica que permita todas las opciones necesarias para la creación de videojuegos del estilo RPG táctico para escenarios isométricos compuestos por bloques. Con estas herramientas combinadas, se dispondría de la creación de mapas, personajes, habilidades y demás utilidades básicas para que un juego funcione correctamente.

2.3.1 - Puntos fuertes

-Está desarrollado por los hermanos Víctor Manuel e Iván José Pérez-Colado y fue presentado como TFG en la Facultad de Informática, teniendo así contacto directo con los desarrolladores (siendo co-director de este trabajo Víctor Manuel) y al código y la memoria del proyecto original de *IsoUnity*.

-En la memoria de IsoUnity se referencia que, Final Fantasy Tactics (y los juegos TRPG en general) fue una de las principales fuentes de inspiración para las decisiones tomadas en la realización del editor de mapas y decoraciones. Este hecho favorece nuestro proyecto al complementarse con el proyecto inicial.

-Su editor permite crear mapas isométricos sin necesidad de tocar ningún tipo de código. Todo se realiza mediante menús y editores gráficos de Unity. Todo este código puede servir de ejemplo para el desarrollo de los nuevos editores pues la documentación sobre el editor de Unity es la más escasa y compleja.

-Los mapas isométricos se dividen en divisiones cuadradas (*celdas*) que pueden tener distintas alturas e inclinaciones dando la posibilidad de crear muros y rampas.

-Las *celdas* pueden ser pintadas usando una base de datos de texturas (*IsoTextures*) especialmente diseñadas para *IsoUnity*.

-El mapa se puede adornar con elementos estáticos (*IsoDecorations*) y dinámicos (*Entidades*).

-Las *entidades* son objetos vivos (pueden hacer cosas) y extensibles (se pueden ir creando componentes que se les agreguen para hacer más cosas).

-El *sistema de secuencias* permite dirigir el flujo de acciones tras la interacción del jugador con una entidad, proporcionando un sistema de programación visual basado en nodos y eventos completamente extensible.

2.3.2 - Puntos débiles

-Aunque *IsoUnity* es completamente funcional es software universitario en desarrollo, no un producto comercial, por lo que nos podremos encontrar limitados por ciertos aspectos, ya sea en la integración o en el propio uso del armazón.

-Dispone de un sistema de inventario, pero no de la manera deseada, ya que no se pueden almacenar los objetos sino que al encontrar objetos, estos ejecutan secuencias.

-Las entidades de IsoUnity necesitan ser ampliadas para soportar personajes con fichas propias de los juegos de rol.

-Aunque hay bases de datos (como por ejemplo para guardar *IsoTextures*) no hay una base de datos genérica que permita guardar persistentemente las fichas de personaje ni todas las variables que intervienen en un juego de rol.

-El sistema de secuencias no es lo más ideal para crear un sistema de batalla, con lo que será necesario crear un sistema completamente aparte y controlado por turnos.

-El mapa de IsoUnity no puede ser muy grande, ya que está siempre en memoria, lo cual limita el tamaño de las escenas de batalla del futuro juego.

Capítulo 3: Objetivos y especificación de requisitos

El propósito general del proyecto es avanzar en el desarrollo de herramientas pensadas especialmente para creadores independientes familiarizados con *Unity* que faciliten el desarrollo de videojuegos del género RPG táctico por turnos y en perspectiva isométrica, todo ello partiendo de la base que proporciona *IsoUnity*.

En este tipo de juegos, el jugador controla a un grupo de personajes que ocupan las casillas cuadradas en las que está dividido el escenario y que pueden moverse libremente por allí, pudiendo realizar acciones sencillas y entrar en un determinado momento en combate contra otro grupo de personajes controlados por el ordenador. Estos combates se realizan por turnos y como resultado de la batalla se obtiene la victoria o la derrota

3.1 - Objetivos

Como objetivos principales del proyecto hemos destacado los siguientes:

- Estudio de los videojuegos de rol táctico y de los videojuegos con estética retro y perspectiva isométrica.
- Estudio de las herramientas de desarrollo de videojuegos más adecuadas para los videojuegos anteriormente mencionados.
- Aprendizaje de la herramienta IsoUnity y de sus posibilidades actuales y facilidades de extensión.
- Extensión de IsoUnity mediante el desarrollo de un sistema de edición de personajes, habilidades y objetos propio de los videojuegos de rol.
- Extensión de IsoUnity mediante el desarrollo de un sistema de combate táctico por turnos.
- Desarrollo de un prototipo jugable que sirva como demostración de las extensiones para videojuegos de rol táctico de estética retro e isométrica que habrán sido desarrolladas.

3.2 - Plan de trabajo

El proyecto comienza con una fase de análisis y estudio de todo lo que rodea al trabajo. Se trata de obtener conocimientos de lo existente actualmente en relación a la cuestión además de analizar lo más utilizado en este tipo de proyectos para

conseguir el mejor resultado posible y obtener conclusiones en las que centrar la atención.

Durante el desarrollo de todo el proyecto se van a realizar cambios a medida que aumente la experiencia de los integrantes del grupo y vayan implementado todo lo mencionado en los objetivos. Para ello, a pesar de comenzar con el ideal de seguir un modelo de **desarrollo en cascada** la realidad ha aconsejado acercar un poco más el desarrollo a un modelo **iterativo e incremental**. Es decir, la idea no es cumplir los objetivos secuencialmente de la *mejor forma* posible sino conseguirlos secuencialmente de la *forma más simple* e iterar sobre ellos para mejorar. Por ello, a medida que los integrantes del proyecto vayan adquiriendo experiencia en el manejo de las diferentes herramientas estudiadas, se realizará una iteración a todo el trabajo realizado y se irán añadiendo todos los conocimientos adquiridos hasta esta iteración. Con ello se consigue que todo lo trabajado anteriormente quede perfeccionado y se puedan ir añadiendo nuevas ideas con lo aprendido en cada iteración.

Este desarrollo iterativo e incremental ha sido enfocado desde un punto de vista ágil, destacando la creación de prototipos para editores y pipelines que más adelante sufrieron cambios hasta llegar a la versión final.

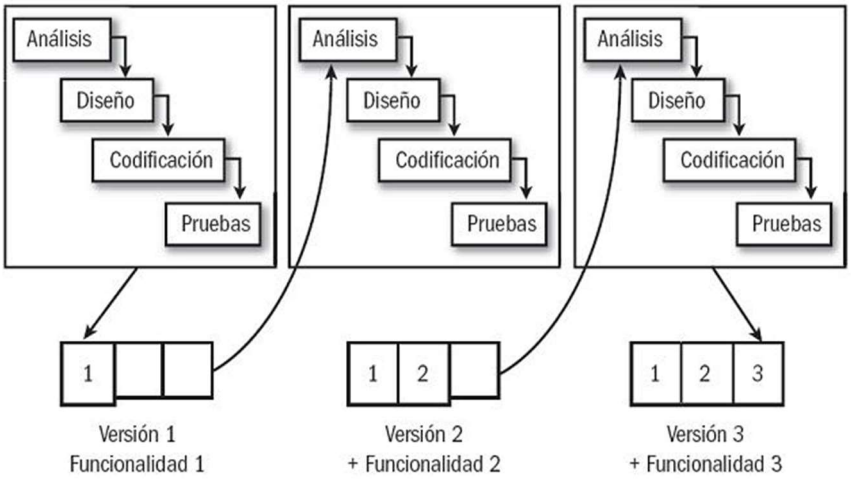


Figura 3.2.1 - Desarrollo iterativo e incremental

Los integrantes irán aprendiendo *Unity* y su funcionamiento, ya que no son expertos en esta plataforma y necesitan cierta formación. Se deben conocer todos los detalles, debilidades y los puntos fuertes de la plataforma para aprovechar al máximo el potencial de éste entorno de desarrollo de videojuegos.

Como recursos para la implementación, ambos autores de este trabajo son los desarrolladores del software y de algunos de los recursos audiovisuales que serán necesarios. Otros serán obtenidos de bibliotecas de libre acceso o serán proporcionados por *IsoUnity* [https://github.com/Victorma/IsoUnity], herramienta sobre la que se fundamenta el desarrollo del proyecto. Como recursos físicos se utilizarán ordenadores con el sistema operativo Windows con componentes capaces de trabajar con Unity y VisualStudio.

Por último, y a medida que se vaya desarrollando y mejorando el proyecto, los desarrolladores tendrán a su disposición un repositorio en GitHub, el sistema para control de versiones basado en Git, donde trabajarán sobre *TRPGMaker* [https://github.com/Narratech/TRPGMaker] y de donde desarrolladores futuros podrán descargar la herramienta y plantear cuestiones sobre ella.

Para el desarrollo de la memoria el proyecto utilizará el sistema de Google Drive ya que permite la edición simultánea de varias personas a la vez, y finalmente se realizará una exportación a PDF para la entrega.

3.2.1 - Estimación del esfuerzo

Cómo el proyecto se irá modificando a medida que se vaya desarrollando, se han seleccionado cinco hitos importantes para estimar el esfuerzo necesario, intentando desarrollar los dos primeros en paralelo (sistema de combate y base de datos) para acabar trabajando con la unión de ellos en el tercero. El cuarto hito será una demo que demuestre la funcionalidad del desarrollo y el quinto y final, la redacción completa de la memoria.

Partiendo de la base de que el TFG son 12 créditos ECTS y cada crédito equivale a 25 horas de trabajo personal la estimación final son de 12 créditos * 25 horas * 2 integrantes = 600 horas totales de trabajo.

El primer hito, que es el que más directamente aborda el objetivo final del proyecto, será el desarrollo de la herramienta que permita implementar **sistemas de combate** de manera genérica (gestión de turno, lanzamiento de habilidades, ataques, defensa…). A priori esta será una gran carga de trabajo ya que no existe nada implementado para el combate desde lo que se pueda empezar. Sin embargo será la parte que explotará IsoUnity para la gestión del mapa y las entidades que se muestran sobre él. El sistema de combate debería incluir el sistema de inventario, ya que dependiendo de los objetos obtenidos el combate, éste se podría desarrollar de diferentes maneras. Estimamos un **30%** de carga de trabajo (**180 horas** del total de 600 horas).

El segundo hito es la creación y gestión de las **bases de datos del juego**. Un sistema de combate necesita de muchos elementos para funcionar (armas, personajes, hechizos, estadísticas de todos estos elementos, cambios de estas estadísticas según las interacciones…). Para el futuro desarrollador de TRPGMaker se antoja una tarea tediosa el gestionar todos estos elementos si la herramienta no da facilidades. Por tanto la base de datos albergará y relacionará estos elementos. El diseño y la implementación de ésta se llevará gran parte del tiempo de trabajo, ya que es más que probable que haya que redefinirla en cada iteración. Incluye además los editores para facilitar la tarea de ingesta de datos al desarrollador. Estimamos un **40%** de carga de trabajo (**240 horas** del total de horas).

El tercer hito consistirá en **relacionar y actualizar los datos en la base de datos con los cambios que se produzcan en el juego durante el sistema de combate**. Será la parte que menos tiempo lleve si los dos primeros hitos se han completado pensando en esta integración, aunque inevitablemente habrá que dedicar cierto esfuerzo a resolverlo. Estimamos un **8%** de carga de trabajo (**48 horas** del total de 600 horas).

El cuarto hito será una **demo jugable** que muestre la capacidad de la base de datos, el sistema de combate y su interacción. Es la parte que demuestra todo el trabajo anterior y debería ocupar un tiempo razonable pero no excesivo al final del desarrollo de la herramienta, ya que no se pretende realizar un videojuego completo, que sería muy costoso, sino una prueba de concepto que demuestre que

efectivamente el sistema es capaz de realizar lo que dice su especificación. Un **8%** del trabajo se lo llevará esta demo (**48 horas** del total de 600 horas).

El quinto hito y final, será la memoria, la cual se irá completando en paralelo al trabajo realizado en la medida de lo posible y ocupará parte del tiempo al principio para el estudio, estimación y planificación del proyecto, y parte al final junto con el desarrollo de la demo antes de estar completa. Estimamos un **14%** aquí (**84 horas** del total de 600 horas).

3.2.2 - Comprobación de la estimación del esfuerzo

Finalmente, el desarrollo del proyecto se ha llevado a cabo durante un curso lectivo completo, desde el inicio de éste hasta las fechas finales de entrega en septiembre. Durante todo el curso se han ido desarrollando todos los requisitos del proyecto, centrándose en el estudio, planificación y diseño la primera parte del curso y en la segunda mitad en la implementación, iteración y mejora del código del proyecto tal como había sido planeado.

Por motivos de estudio, coincidencia de exámenes y esfuerzo con otras asignaturas ha habido algunas fechas en las que no se ha podido trabajar tanto como se desearía en el proyecto, sobre todo durante el primer cuatrimestre, pero esas horas han sido recuperadas después para ofrecer el producto completo y finalizado.

Para confirmar los cálculos de estimación de esfuerzo previstos en la sección 3.2.1 se han ido rellenando unas tablas mes a mes con las horas consumidas en trabajar en las distintas tareas planteadas (Figura 3.2.2.1). El rellenado de estas tablas se ha hecho en una hoja de cálculo que genera un gráfico de consumo de tareas (*burndown chart*) en tiempo real. Eso nos ha permitido comprobar en el momento si íbamos retrasados o adelantados respecto al trabajo estimado.

Mes	Burned down (h)		Restante (h)		Hecho
	Planeado	Real	Planeado	Real	
sep-16	0	0	600	600	0
oct-16	50	10	550	590	10
nov-16	50	15	500	575	15
dic-16	50	12	450	563	12
ene-17	50	2	400	561	2
feb-17	50	55	350	506	55
mar-17	50	95	300	411	95
abr-17	50	105	250	306	105
may-17	50	80	200	226	80
jun-17	50	21	150	205	21
jul-17	50	60	100	145	60
ago-17	50	60	50	85	60
sep-17	50	36	0	49	36

Horas planeadas	Tarea
600	Total
180	Sistema de combate
240	Bases de datos
48	Relacionar combate y BDs
48	Demo jugable
84	Memoria

Figura 3.2.2.1 Tablas que se fueron rellenando durante el año para calcular el gráfico de burndown a partir de ellas

Viéndo el *gráfico de burndown* generado (Figura 3.2.2.2) se confirma lo dicho anteriormente: en el primer cuatrimestre apenas se avanzó (sobre todo se hizo en temas de memoria y documentación) para parar casi al completo durante los exámenes de enero y febrero. En el segundo cuatrimestre se trató de recuperar el tiempo perdido llegando cerca de la entrega a finales de mayo (llegamos a tener una demo rudimentaria pero la memoria estaba francamente incompleta). Hubo que posponer la entrega a septiembre. Junio fue un mes calmado coincidiendo con exámenes. Julio y agosto se dedicaron a pulir la memoria y la demo final. Al final se hicieron un total aproximado de **551 horas** ya que no siempre se llevó el registro de trabajo de manera exhaustiva.

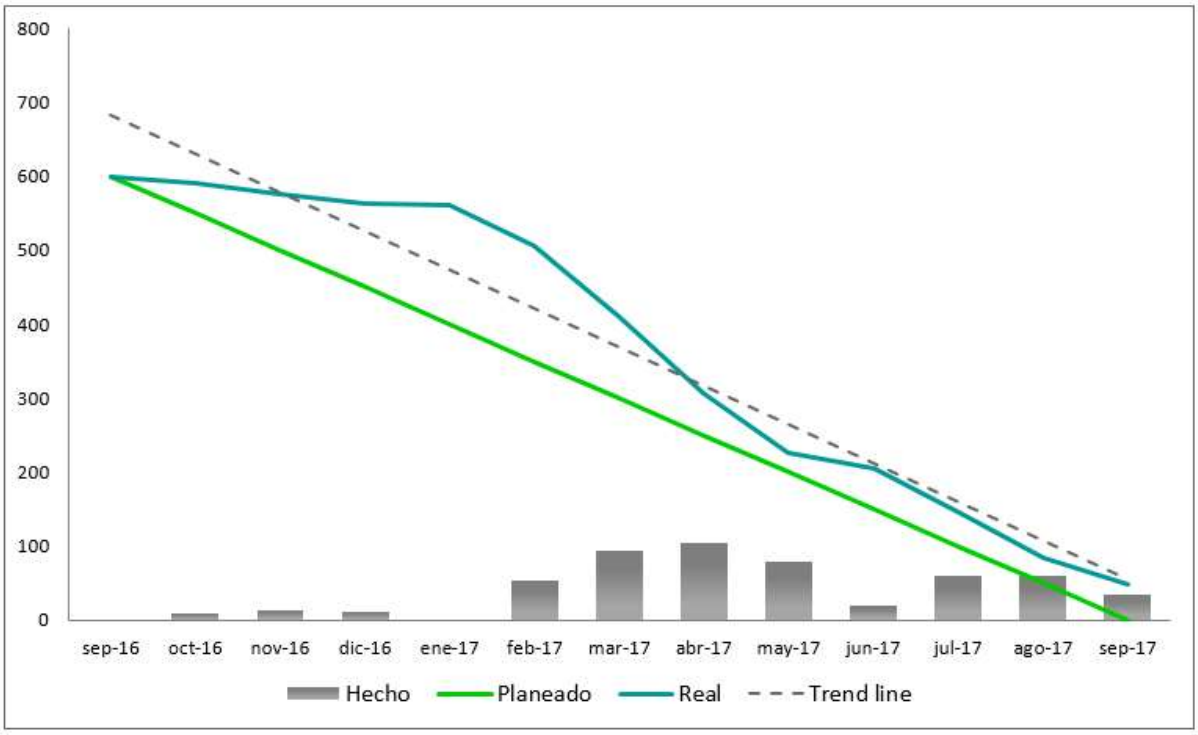


Figura 3.2.2.2 Gráfico de burndown que muestra el desarrollo del trabajo durante el curso lectivo

Cabe destacar que al ser una herramienta tan abierta, TRPGMaker siempre se podrá ampliar y mejorar después de la entrega final, ya que aunque lo entregado sea funcional completamente (v1.0) está a nivel de una versión beta. En un futuro se deberían pulir las funcionalidades actuales y se podrán agregar nuevas utilidades: TRPGMaker se ha dejado preparado para ello.

3.3 - Especificación de requisitos software

En los siguientes subcapítulos se especifican los requisitos software para los objetivos planteados.

3.3.1 - Elementos y parámetros necesarios para un TRPG

En cualquier RPG (TRPGs inclusive) el mundo del juego está poblado de *elementos* (personajes, enemigos, objetos…). Estos elementos se caracterizan por

compartir seguir un esquema común entre sí, que permite al sistema realizar de forma sistemática las mecánicas, acciones y consecuencias. Por ello, como pudimos ver en RPG Maker, los motores RPG suelen contar con una base de datos común que define todas las reglas y elementos que participan en el mundo de juego. Esto incluye una definición de propiedades y parámetros comunes, que intervienen en el cálculo de las estadísticas del personaje y por lo tanto, sus capacidades para ejercer daño, defenderse, esquivar u otros resultados.

Por ello, a continuación, planteamos un sistema genérico que permite definir atributos y tags para establecer características y fórmulas para establecer sus relaciones y sus elementos consecuentes, los objetos y las pasivas. A continuación, un elemento definitorio de los RPG es su inventario, formado por los diferentes espacios donde almacenar objetos y equiparlos en base a su tipo. Para ello, introduciremos los slots, que permiten establecer en ellos todos los objetos, pasivas e incluso clases, que, en conjunto, describirán cada personaje.

3.3.1.1 - Los atributos (*attributes*)

A lo largo del análisis del estado del arte hemos podido observar la cantidad y diversidad de parámetros que se utilizan en los juegos. Pese a que existen parámetros clásicos como el ataque, la defensa, la resistencia a la magia o resistencias a los diferentes elementos, también existen parámetros muy específicos para cada juego como la popularidad en Chroma Squad y el combo en FFT. Todos ellos, aún así, disponen de las mismas capacidades para describir objetos, relaciones, interacciones, etc.

Los atributos son entonces una propiedad añadida a cualquier elemento del juego que cuenta con un valor numérico. Con el objetivo de simplificar la labor de autoría, contaremos con un editor que dará la posibilidad de crearlos. Sin embargo no todos los atributos son iguales. Deberíamos definir los *atributos básicos (basic attributes)* como aquellos que existen en cualquier RPG siendo imprescindibles para cualquier personaje. Son la experiencia (EXP) para evidenciar cómo evoluciona el personaje, la vida o health points (HPS) y la magia o magic points (MPS) que permite realizar habilidades especiales.

Después de éstos estarían los *atributos centrales (core attributes)*. Son los que define el diseñador del juego. Todos los personajes tienen unos atributos centrales de base, cuyos valores pueden ser modificados permanentemente según el jugador obtenga experiencia (EXP) o temporalmente según lo que ocurra en el juego. Definen las cualidades del personaje, dándole más o menos poder en ciertas áreas.

Finalmente los *atributos derivados (derived attributes)*. Estos atributos se generarán a partir de los valores de otros. Por ejemplo el nivel del personaje (lvl) podría generarse a partir de su experiencia (EXP) y la defensa (def) podría generarse a partir de los valores de defensa de los elementos que el personaje lleve equipado y de algún atributo central como podría ser la constitución (CON). El diseñador decide cuáles son los atributos derivados y piensa cuáles van a ser esas relaciones que les dan valor mediante *fórmulas*, sobre las que hablaremos más adelante.

3.3.1.2 - Las etiquetas (*tags*)

Además de los atributos, otro *parámetro* serán las etiquetas. Se usarán para que TRPG Maker pueda relacionar las *clases y especializaciones*, los *ítems* y las *pasivas* con los *personajes*, que también se tratarán a continuación.

Para entenderlo con un ejemplo: un ítem que tenga la etiqueta “two handed” porque necesita ser usado a dos manos, sólo podrá ser usado en los personajes que acepten dichas etiquetas “two handed” para los *ítems* (o explicado de manera más sencilla, todo aquel personaje que tenga una etiqueta “two handed” en sus posibles ítems equipables)

3.3.1.3 - Las fórmulas (*formulas*)

Las fórmulas serán otro *parámetro*: expresiones matemáticas que modificarán los valores de los atributos. Desde valores absolutos simples (suman o restan) pasando por porcentajes (multiplican o dividen) o expresiones más complejas.

Podrán asociarse a *clases y especializaciones*, *ítems*, y *pasivas*. O simplemente existir relacionando dos *atributos*: por ejemplo un atributo derivado como el nivel (lvl) se calculará a partir de uno básico como la experiencia (EXP).

3.3.1.4 - Las ranuras (*slots*)

Las ranuras o slots definen los espacios que cuenta un personaje para poder añadir elementos modificadores en ellos. Estos elementos modificadores se componen de un conjunto de fórmulas y existen tres tipos básicos: clases, ítems y pasivas. Por ello son esenciales a la hora de definir e instanciar el personaje final. Esto será explicado más adelante, con los *personajes*.

Para poder establecer qué elementos pueden incorporarse a una ranura se utilizan los tags, que limitan su contenido. Por ejemplo y de esta forma, un slot de clase puede incorporar otro slot que defina las propiedades raciales de un personaje a traves de un slot con el tag “races” y, por otro lado, la profesión del personaje a través de un slot “professions”.

3.3.1.5 - Las pasivas (*passives*)

Son un *elemento* del juego que complementa las típicas acciones directas que realiza un personaje en un RPG cualquiera (por ejemplo, apretar un botón que ejecutará tal acción que tendrá tal efecto en el combate).

Las *pasivas* son acciones que un personaje ejecuta de forma pasiva. Se definen con un momento dado (permanentemente, al final de un turno, cada equis tiempo…) hacia un objetivo concreto (el mismo personaje, un amigo, un enemigo…). Estas acciones afectan a los *atributos*, por tanto se definen mediante una o varias *fórmulas*.

Por ejemplo, un jugador (o un personaje no jugable) podría tener una pasiva que hiciese un ataque extra al final de cada turno de manera aleatoria. Otro ejemplo sería que un enemigo pudiese tener una pasiva que le permitiese recuperar un 10% de su vida actual cada 30 segundos.

Las *pasivas* tienen la posibilidad de llevar *slots* en las que se pueden asociar más pasivas. Las *ranuras* son un concepto importante que será explicado a continuación.

3.3.1.6 - Los objetos (*items*)

Uno de los *elementos* clave del juego. Los *ítems* no tienen una clasificación concreta. Es el diseñador el que decide las cualidades de un ítem por medio de sus *etiquetas*. Los *ítems* pueden llevar asociadas *fórmulas* que modifican *atributos* de las fichas de los *personajes*.

Como *ranuras* tienen la posibilidad de asociar otros *ítems* y *pasivas*. Por ejemplo podríamos tener un ítem “espada común” con una fórmula que modificase el parámetro ataque (atk) con +10, con una ranura para ítems con los tags “addon” y “gema mágica” y otra que permitiese pasivas que ocurren al final del turno sobre el personaje que equipe el ítem. En la instancia final al crear al personaje con una “espada común” podríamos llenar las ranuras proporcionadas con el ítem y pasiva que creyéramos conveniente (o dejarlas vacías).

3.3.1.7 - Las clases y especializaciones (*classes* & *specializations*)

Las *clases* y las *especializaciones* son las categorías que definen los personajes. Las *clases* son los tipos básicos de personaje. Las *especializaciones* son como su nombre indica especializaciones de las *clases*, es decir, tipos especializados de personaje.

Un ejemplo aclara el concepto: la clase “humano” podría tener distintas especializaciones como “espadachín” o “mago”. La clase “orco” podría tener distintas como “guerrero” o “hechicero”.

En definitiva y al igual que los *ítems*, las *clases* y las *especializaciones* pueden tener *fórmulas* que modifican *atributos*. También pueden tener *ranuras* para *pasivas*, *ítems* y además *especializaciones.* Esto genera un árbol de posibilidades para poder crear un personaje como veremos cuando expliquemos los *personajes*.

3.3.1.8 - Las habilidades (*abilities*)

Las *habilidades* son un tipo especial de comando que se puede ejecutar durante el combate. En general, una habilidad cuenta con un lanzador y con uno o varios receptores, así como una consecuencia del impacto. En contra a los ataques

clásicos, las habilidades permiten dar al desarrollador un toque personal a las acciones que el jugador puede llevar a cabo durante el juego.

Si tomamos una habilidad como un tipo especial de acción, es importante tener un editor rico para definir la forma y consecuencias de la misma. Por ello, se ofrecerán editores para definir el rango y la forma del impacto de una habilidad, así como las diferentes variantes que pueden sufrir en base a la posición del personaje o la cantidad de receptores que una habilidad pueda tener.

Todo juego de estrategia se caracteriza porque cada tipo de unidad cuenta con habilidades únicas y debilidades que se complementan con las otras unidades aliadas. Por ello, se limitará su uso mediante el sistema de requisitos implementado en el desarrollo de la habilidad que será añadido como una extensión del creador de habilidades.

Esta característica añade un nivel estratégico bastante importante a la hora de usar las habilidades ya que cada una es única y solo debe ser usada por algunos elementos.

3.3.1.9 - Los personajes (*characters*)

Los *personajes* son las instancias finales que representan la ficha de personaje. La ficha son todos los datos (atributos, slots y contenido) que necesitaremos para asociarlos a un actor del juego.

Los *atributos* de la ficha se rellenan mediante el **algoritmo de creación de ficha**: el diseñador decide el nombre del personaje, su clase, especialización (o especializaciones), el valor de sus *atributos básicos* y de sus *atributos centrales* y rellena las *ranuras* que han proporcionado la clase y especialización(es) elegidas. La fase de diseño será muy importante en este punto.

Rellenar las *ranuras* con los *ítems* y *pasivas* que se decidan puede a su vez generar nuevas *ranuras*. Rellenar éstas puede generar otras nuevas y repetirse el proceso. Este bucle de generación de nuevas ranuras acaba en el momento que no se rellenen más ranuras o cuando las que se rellenen no generen nuevas.

Una vez se ha estabilizado el relleno de *ranuras* el diseñador seleccionará si los *ítems* que van en ellas están equipados o no. Los que no lo estén se enviarán al inventario. Con las *pasivas* equipadas se actúa según sus parámetros de “cuándo” y “a quién”. A partir de ahí, chequeando las *fórmulas* de todos los elementos que intervienen se sacarán los valores finales de los *atributos* de la ficha.

3.3.2 - Poblar los elementos, parámetros y personajes

Una vez claros los elementos y parámetros necesarios para la creación de este tipo de videojuegos se hace esencial la necesidad de las herramientas que faciliten la tarea crearlos y relacionarlos.

Esto implica la creación de una *base de datos* que guarde las relaciones necesarias y la forma de rellenarla, que será mediante *editores*.

Se hace necesario separar la base de datos en tres para tener modularidad: la *base de datos principal* que gestiona todos los elementos y atributos que sirven para crear personajes finales; la *base de datos de habilidades* que se comienza rellenando con unas habilidades básicas que muestran el funcionamiento de cada tipo de daño, distancia, coste de maná, etc y la *base de datos de personajes* que consulta y elige los elementos y atributos de la base de datos principal para crear personajes.

3.3.2.1 - La base de datos principal y sus editores

A modo de tabla resumen estas son las funcionalidades que deberán tener los editores de los elementos/parámetros y la base de datos principal:

Editor	**Requisitos sobre el elemento o parámetro**
Editor de atributos	-Dar nombre -Dar id (string) -Indicar si es básico, central o derivado -Dar descripción -Dar avatar -Dar valor predeterminado -BD: nuevo, guardar, cargar, editar, eliminar

Editor de etiquetas	-Dar nombre (actúa de id y es autodescriptivo) -BD: nuevo, guardar, cargar, eliminar
Editor de fórmulas generales	-Dar nombre (actúa de id) -Dar descripción -Indicar a qué atributo afecta (mediante su id) -Dar la propia fórmula -Dar la prioridad -BD: nuevo, guardar, cargar, editar, eliminar
Editor de pasivas	-Dar nombre (actúa de id) -Dar descripción -Dar avatar -Dar tags de tipo “cuándo” y de tipo “a quién” -Dar fórmulas y sus prioridades -Permitir slots configurables de tipo pasiva -BD: nuevo, guardar, cargar, editar, eliminar
Editor de objetos	-Dar nombre (actúa de id) -Dar descripción -Dar avatar -Dar tags -Dar fórmulas y sus prioridades -Permitir slots configurables de tipo ítem y de tipo pasiva -BD: nuevo, guardar, cargar, editar, eliminar
Editor de clases y especializaciones	-Dar nombre (actúa de id) -Dar descripción -Dar avatar -Dar fórmulas y sus prioridades -Permitir slots configurables de tipo ítem, de tipo especialización y de tipo pasiva -BD: nuevo, guardar, cargar, editar, eliminar

3.3.2.2 - La base de datos de habilidades y su editor

Estas son las características y funcionalidades que debe ofrecer el editor de habilidades y su base de datos:

Editor	**Requisitos sobre el elemento o parámetro**

Editor de habilidades	-Dar nombre -Dar Descripción -Definir cantidad de Daño -Definir distancia de lanzamiento -Definir tipos de daño, objetivo, habilidad… -Opcional: Añadir requisitos a la habilidad. - Seleccionar tipo de requisito - Definir el requisito seleccionado

3.3.2.3 - La base de datos de personajes y su editor

A modo de tabla resumen estas son las funcionalidades que deberá tener el editor de los personajes y la base de datos de los personajes:

Editor	**Requisitos sobre el elemento o parámetro**
Editor de personajes	-Dar nombre (actúa de id) -Dar descripción -Dar avatar -Dar valor de clase y de especialización(es) -Dar valores a los atributos básicos y centrales -Generación de slots de ítems y pasivas -Dar valor a los slots de ítems y pasivas -Decidir qué ítems de los puestos en los slots van equipados -BD: nuevo, guardar, cargar, editar, eliminar

3.3.3 - Sistema de combate táctico

Para el modelo de combate se han seleccionado los estilos más utilizados por otras herramientas y juegos y se ha implementado un sistema de combate por turnos basado en la velocidad de los personajes. Así a cada personaje se le otorgará el turno en función de su parámetro “speed” siendo el personaje más rápido el primero en jugar y el más lento el último.

Cada comando del sistema solo se puede usar una vez en cada turno, por lo que obliga al jugador a plantear correctamente como tiene que ejecutar sus movimientos y habilidades para lograr el objetivo de la misión.

En el turno de cada personaje se dispondrá de cuatro opciones, Attack, Use Skill, Move y Defend.

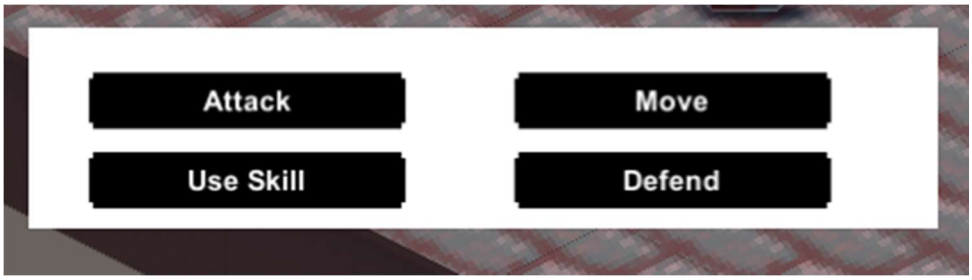


Figura 3.3.3.1 Acciones durante un turno

Al usar "**Attack**" el jugador seleccionará el objetivo enemigo al que quiere realizar el ataque y si está al alcance de su arma principal, podrá realizarle daño básico (basado en el daño de su arma y su fuerza). Usar esta función provoca que la segunda opción de realizar daño se cancele ("Use Skill") ya que queremos que solo se pueda realizar una acción de daño por turno.

Al usar "**Use Skill**", aparecerá una lista de habilidades disponibles para el personaje, basándose en los requisitos de la habilidad y el maná restante del personaje. Cada habilidad tiene un alcance definido por el desarrollador y realizará un efecto único. Recordar que el objetivo debe estar al alcance de la habilidad o no se podrá seleccionar ningún enemigo como objetivo. Usar una habilidad elimina la posibilidad de usar el comando Attack durante este turno.

Figura 3.3.3.2 Habilidades durante el juego

Al usar el comando “**Move**” el jugador podrá mover a su personaje a otra casilla del mapa dependiendo de la velocidad del personaje, cuanto más veloz sea, más rápido podrá moverse y su movimiento será más amplio, teniendo acceso a casillas más lejanas. Es importante que el desarrollador implemente sus personajes con lógica para que el jugador tenga una buena experiencia y no piense que los personajes sean ilógicos (un arquero debería poder atacar o moverse más rápido que un caballero, por ejemplo).

Para terminar el turno, el jugador puede hacer uso en cualquier momento de la opción “**Defend**” que provoca que el personaje le diga al sistema que ha terminado su turno y da paso al siguiente jugador. Este comando se puede utilizar a lo largo de todo el turno del jugador, y no es necesario que haya realizado las acciones anteriores para dar paso al siguiente personaje.

3.3.4 - Requisitos software y audiovisuales para la demo

Los requisitos software para la demo son los que proporciona TRPG Maker sobre IsoUnity: un sistema de creación de elementos del juego, un sistema de batallas y un editor de escenarios. Los requisitos audiovisuales serán spritesheets para los personajes, sprites para decorar el escenario, texturas para los tiles del escenario, música de fondo y alguna imagen para ilustrar el fondo. Recursos extras

como tipografías o diseños para menús ingame también podrían ser necesarios. Será en la fase de diseño cuando se concreten estos elementos.

Capítulo 4: Análisis y diseño

En este capítulo trataremos el análisis y el diseño del sistema (y del correspondiente prototipo de juego ilustrativo) especificados en el capítulo anterior. Para ello dividiremos la tarea en distintos apartados, de los cuales hablaremos de su análisis y diseño individualmente:

1. Arquitectura global del sistema
2. Las bases de datos y la carga de datos al juego
3. Sistema de combate táctico y la conexión con IsoUnity
4. Juego demostrativo

4.1 - Arquitectura global del sistema

El sistema de TRPG Maker funcionará como una extensión de Unity, por lo que no tiene mayor complicación que incluir una carpeta con su contenido en el directorio raíz del proyecto que hayamos creado en Unity para disponer de todas las herramientas y utilidades de TRPG (Figura 4.1.1).

Sin embargo TRPG Maker funciona también como un complemento de IsoUnity ya que utiliza sus recursos de creación y gestión de mapas isométricos. Por tanto el sistema debe incluir también una implementación concreta de IsoUnity en el directorio raíz del proyecto (Figura 4.1.1). Esta relación de dependencia no impide en ningún caso desarrollar IsoUnity por un lado y TRPG Maker por otro, facilitando la modularidad del sistema. De hecho cambiar la implementación de IsoUnity es un proceso transparente para el desarrollador de TRPG Maker (actualizar los contenidos de la carpeta IsoUnity).

Figura 4.1.1 Carpetas de IsoUnity y TRPGMaker en la carpeta raíz del proyecto de Unity

Analizando la estructura de carpetas de IsoUnity es aconsejable que TRPG Maker siga una nomenclatura similar para facilitar el trabajo del desarrollador (Figura 4.1.2).

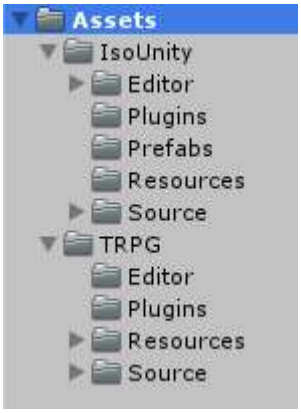


Figura 4.1.2 Estructura de las carpetas de IsoUnity y TRPG Maker en la carpeta raíz del proyecto de Unity

En concreto TRPG Maker va a necesitar usar los elementos Game y Map de IsoUnity para poder generar una escena para una batalla táctica por turnos. La Figura 4.1.3 muestra exactamente eso: que IsoUnity es un sistema independiente creado sobre Unity y TRPG Maker es un sistema dependiente de IsoUnity creado también sobre Unity. Para nuestro caso concreto IsoUnity necesitará también las actualizaciones de la batalla para poder actualizar la visualización del juego.

Figura 4.1.3 Relación entre IsoUnity, TRPG Maker y Unity

A su vez TRPG Maker va a estar dividido en dos módulos diferenciados que corresponden a las bases de datos por un lado y el sistema de batalla por otro, los cuales son interdependientes (El sistema de batalla necesita la información de los personajes creados en la base de datos de personajes y la base de datos de los personajes necesita la actualización de los atributos que se producen por las acciones del juego durante la batalla) como vemos en la Figura 4.1.4.

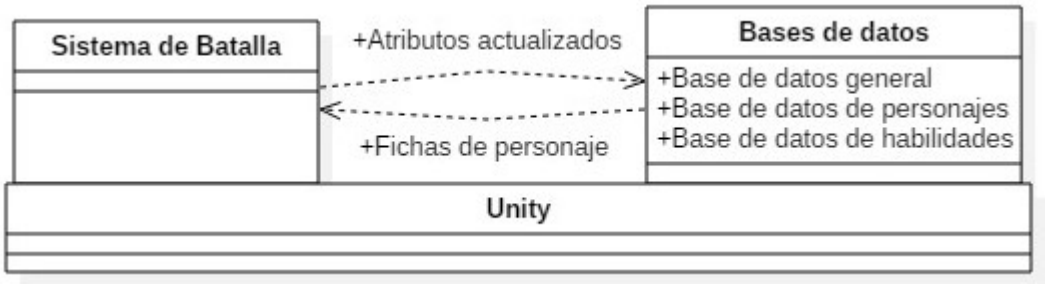


Figura 4.1.4 Relación y dependencias entre la base de datos de personajes y el sistema de batalla dentro de TRPG Maker

Esos mismos módulos van a tener también una interdependencia que se produce con la base de datos de habilidades, la cual se debe consultar desde el sistema de batalla para lanzar habilidades y que permite desbloquear nuevas habilidades en la base de datos según lo que ocurre en la batalla del juego (Figura 4.1.5).

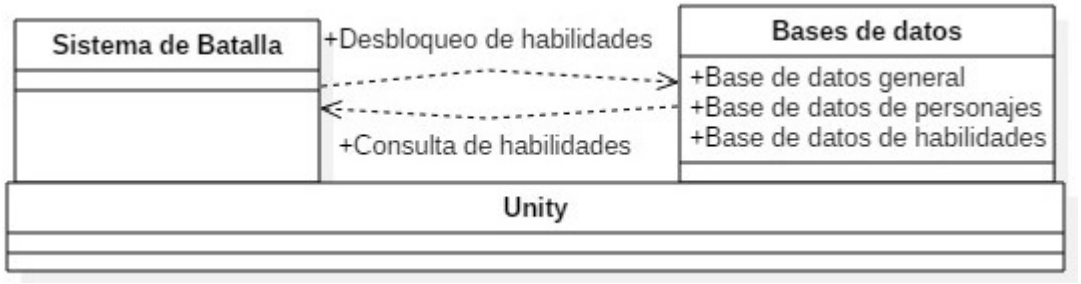


Figura 4.1.5 Relación y dependencias entre la base de datos de habilidades y el sistema de batalla dentro de TRPG Maker

Por tanto la relación final de los distintos módulos de TRPG Maker e IsoUnity se contempla en la Figura 4.1.6 con las interdependencias simplificadas.

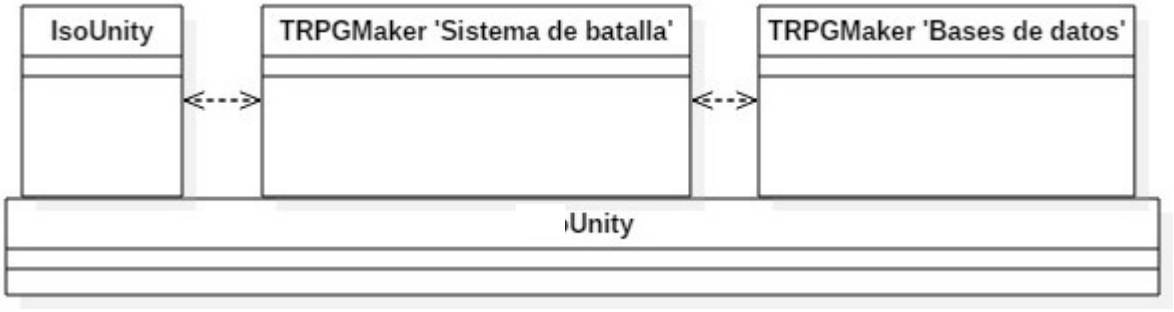


Figura 4.1.6 Esquema final simplificado de la arquitectura

4.2 - Las bases de datos y la carga de datos al juego

Los elementos creados a lo largo del desarrollo del juego o del mapa del juego se deberían almacenar en archivos. Si estos archivos se guardasen en un formato entendible por Unity los llamaríamos *assets* que es como se designa comúnmente a cualquier elemento de un juego dentro de Unity. También podrían ser guardados en formato de texto o binario.

La opción de crear assets se integra mejor con Unity pero requiere de más estudio de la propia documentación de los assets en Unity. La segunda no se integra tan bien con Unity pero es una aproximación simple sin apenas curva de aprendizaje. En cualquier caso almacenar los elementos del juego con sus correspondientes atributos y relaciones es algo obligatorio para tener persistencia permitiendo así que el desarrollo de juegos en TRPGMaker no se tenga que hacer en una sola sesión y repitiendo tareas una y otra vez.

Deberíamos distinguir cuatro bases de datos (en forma de documentos o assets) que se cargasen al iniciar el diseño de un juego sobre TRPGMaker.

El **primer documento almacenaría toda la información sobre el mapa, las decoraciones y todo lo relacionado con lo visual**. La disposición de los objetos y personajes en las diferentes casillas, las decoraciones de estas e incluso los objetos que se colocan en una celda. Este archivo debería ser claramente mantenido y generado por IsoUnity.

El **segundo asset debería ser una gran base de datos que almacenase toda la información requerida para que el sistema de rol funcionase**: clases, equipamientos, niveles y más información general se almacenaría aquí. En resumen, este archivo contendría toda la información necesaria para que la característica de rol del proyecto funcione ya que sin esta base de datos no se podría conocer el estado de cada personaje o elemento activo del juego.

El **tercer archivo almacenaría toda la información de las habilidades**: cualquier habilidad creada sería almacenada para después ser cargada al inicio de proyecto, así el programador/diseñador no debería preocuparse por crear siempre las habilidades. Destacar que todos los cambios en las habilidades se deberían poder hacer en tiempo real y si se están ejecutando el juego, al borrar o crear una habilidad aparecería como disponible para los personajes que cumplan sus requisitos. Este archivo, al igual que los mencionados anteriormente se actualizaría a medida que se realizan cambios y siempre se cargase con el arranque del proyecto que contenga a TRPGMaker.

El **cuarto asset almacenaría los personajes en sí**. Los personajes deberían ser entidades que tuviesen definidas como propiedades propias algunas de las especificadas en el segundo asset (atributos, items…).

4.2.1 - Base de datos general, clases y editores

Para diseñar este asset se ha tenido que pensar cómo encajar todas las piezas que componen los elementos de un RPG a la vez. Esta ha sido quizás la tarea de análisis y diseño más difícil en cuanto a las bases de datos.

Si bien en la especificación (apartado 3.) quedó claro que los elementos para definir un juego de rol son *atributos, etiquetas, fórmulas, ranuras, pasivas, objetos, clases y especializaciones, habilidades,* y *personajes* lo que no estaba claro es cómo iban a interactuar unos con otros. Como hemos dicho que las *habilidades* se consideran en una base de datos independiente y los *personajes* también (dependen siempre de la instancia del resto de elementos definidos) se hace evidente que la base de datos general contendrá la forma de guardar de forma permanente *atributos, etiquetas, fórmulas, ranuras, pasivas, objetos, clases* y *especializaciones*.

4.2.1.1 - Base de datos general (singleton)

Cuando esta base de datos general sea implementada debería ser solamente una. Usar un patrón **singleton** (una sola instancia de un objeto accesible desde cualquier parte del código en ejecución de TRPGMaker) ayudaría a la comprensión y seguridad. Además actuará como controlador decidiendo qué entra y qué no en sus estructuras.

4.2.1.2 - Clases para los elementos y sus relaciones

Debería existir **una clase para los atributos**, **otra para las etiquetas** y **otra para las fórmulas**. Los **objetos**, las **clases y especializaciones** y las **pasivas** deberían tener también **cada una su respectiva clase**. El singleton a su vez tendría guardados conjuntos de esas clases (representados en la Figura 4.2.1.2.1 con la flecha acabada en rombo negro, también llamada *composición*).

Además las **clases** de **objetos**, **clases y especializaciones** y **pasivas** especializan una clase para un **elemento genérico** (representado en la Figura 4.2.1.2.1 con flechas acabadas en triángulo blanco, también llamada *generalización*).

Por otro lado, las clases de **objetos**, **clases y especializaciones** y **pasivas** tienen ranuras que se pueden rellenar. Se necesita alguna forma de decir qué puede ir en esos slots y por eso todas esas clases llevan asociada una clase de **configuración de slots** con una relación de composición.

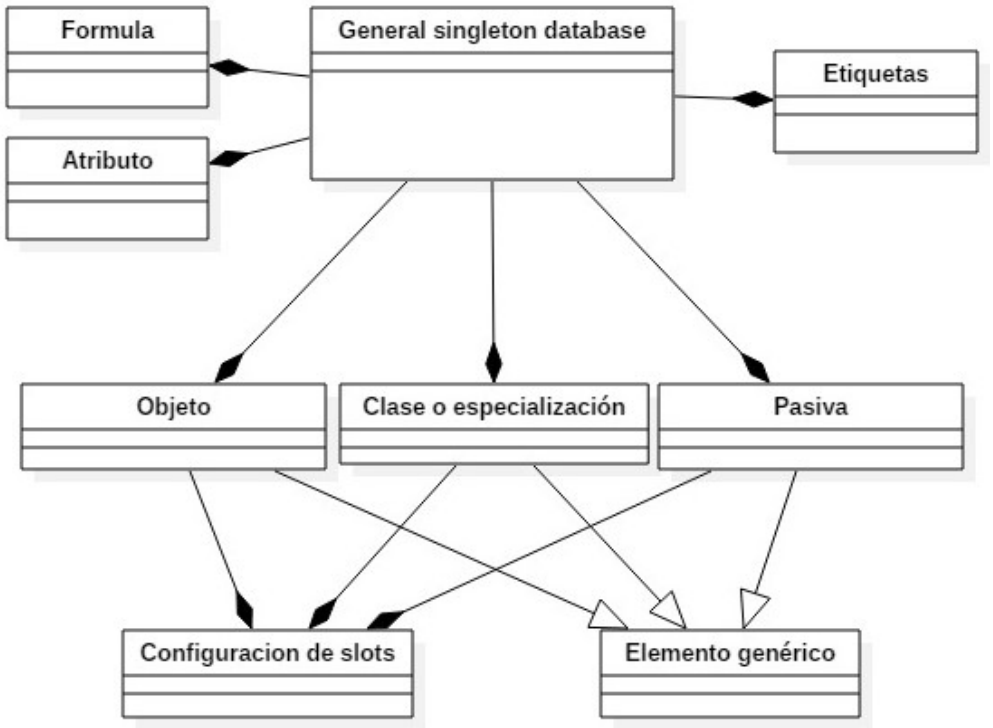
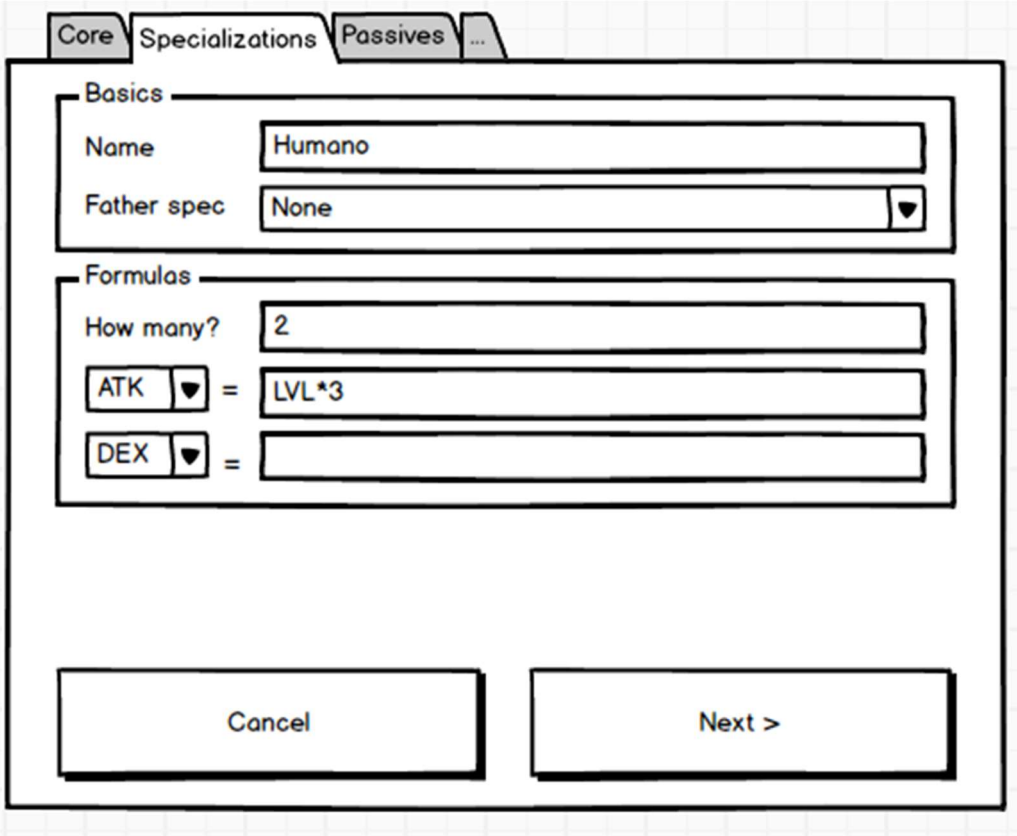


Figura 4.2.1.2.1 Diseño de alto nivel de las clases de la base de datos general (las fechas blancas indican herencia y los rombos negros composición)

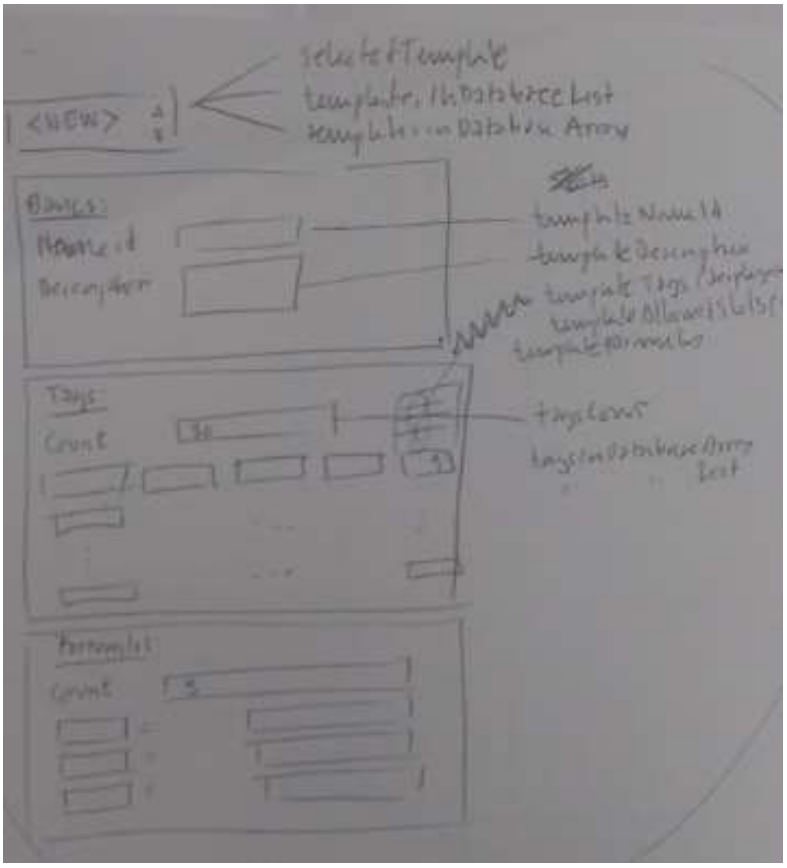
4.2.1.3 - Editores para los elementos y su relación con las clases

Evidentemente las instancias de estas clases van a necesitar ser creadas y pobladas de alguna manera y es ahí donde se materializa la necesidad de editores de *atributos*, *etiquetas*, *fórmulas*, *objetos*, *pasivas*, y *clases y especializaciones*. Los *slots* no tendrían editor propio, sino que iría incrustado dentro del de *objetos*, *pasivas* y *clases y especializaciones*.

La figura 4.2.1.3.1 muestra un mockup para un wizard (típicos menús de siguiente, siguiente…) de población de una *clase o especialización*. En ese paso del wizard se pueden actualizar valores básicos y fórmulas. Pulsando en *Next* seguiría dando opciones, como por ejemplo rellenar *slots* o configurar *etiquetas*. Después de varias iteraciones con mockups como este y con diseños en papel (Figura 4.2.1.3.2) llegamos a los editores finales que se verán en el Capítulo 5 ya implementados.

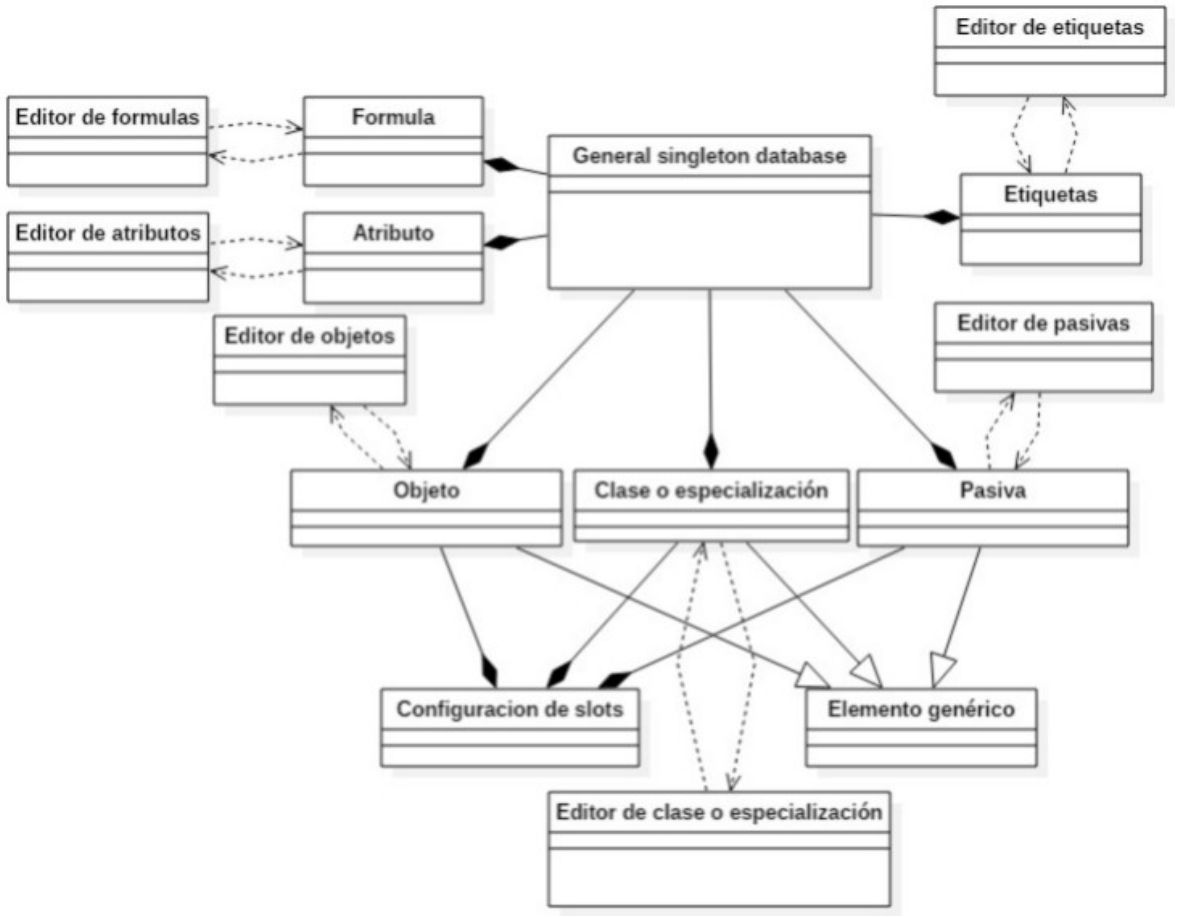


4.2.1.3.1 Mockup en formato wizard para las clases y especializaciones



4.2.1.3.2 Diseño en papel para el elemento genérico

La relación entre los editores y las clases de cada elemento se aprecia en la figura 4.2.1.3.3. Hay una interdependencia (línea discontinua en ambos sentidos) que identifica que para crear elementos de un tipo hace falta un editor concreto de ese tipo.



4.2.1.3.3 Diseño de alto nivel de dependencias de clases de objetos y sus editores

4.2.2 - Base de datos de habilidades, clases y editores

La herramienta desarrollada ofrece una utilidad para la creación de habilidades que los personajes pueden utilizar a lo largo de los diferentes mapas existentes en el juego. Para ello, el desarrollador debe acceder al menú habilidades que hay dentro de las herramientas de TRPGMaker.

Allí se permite la creación de las habilidades, para lo cual se especifica un nombre y una descripción y diferentes variables que permiten al motor calcular el tipo de daño, el efecto y el funcionamiento de la habilidad creada y está habilidad ya será funcional en el juego. El sistema está programado en tiempo real por lo que cualquier habilidad creada durante el juego se podrá acceder y cargar en el

siguiente turno del jugador. Para definir su uso en la arquitectura es necesario definir la forma en la que las habilidades actuarán al ser utilizadas. Para ello proponemos tres tipos de lanzamientos: singulares, área o globales. Los lanzamientos singulares sólo golpearán a un enemigo mientras que, los lanzamientos en área ejecutarán su efecto sobre todos los personajes dentro de un área. Por último, los lanzamientos globales afectarán a todos los enemigos sin tener que definir un rango o área. Dentro del efecto, además se puede definir quienes sufren las consecuencias, pudiendo filtrar entre aliados, enemigos u ambos. Finalmente, el rango define la distancia máxima sobre la cual se puede lanzar una habilidad.

Las habilidades disponen de un sistema de Requisitos que los personajes deben de cumplir para poder usarlas. Para ello se debe especificar qué tipo de "requisito" se quiere añadir a la habilidad. Por ejemplo, se puede hacer uso de requisitos "nivel" que permite delimitar una habilidad a personajes que tengan un nivel específico. Aclarar que no es necesario hacer uso de este sistema de requisitos y las habilidades se pueden crear sin éstos.

Aunque las habilidades actualmente no disponen de animaciones, se quiere agregar al editor un sistema o menú que permita generar diferentes animaciones para las habilidades creadas por el desarrollador, para que el personaje que utilice dicha habilidad genere una serie de movimientos que "identifiquen" que el personaje está usando una habilidad (por ejemplo: que una habilidad del tipo fuego, el personaje que utilice la habilidad lance una llama).

4.2.3 - Base de datos de personajes, clases y editor

Para diseñar este asset se ha tenido que pensar cómo será el algoritmo de creación de ficha de personaje. Tras varias iteraciones en las que no se tenía muy claro cómo se iban a generar los slots finales para que pudiesen rellenarse se llegó a la conclusión de que el algoritmo siguiese los pasos indicados en la tabla 4.2.3.1.

Rellenado inicial: se rellenan distintos parámetros de forma secuencial	
1- Rellenar datos básicos	Nombre (id) y descripción

2- Rellenar atributos básicos y centrales con valores iniciales	Atributos básicos: experiencia, health points, magic points. Atributos centrales: decididos a la hora de crear atributos en su editor
3- Elegir clase	Al elegir una clase se deben ofertar las posibles especializaciones que tiene
4- Elegir especialización	Al elegir una especialización el algoritmo entra en modo iteración
Modo iteración: se comprueban los elementos que generan la clase y especialización elegidas y se itera repitiendo el mismo proceso sobre los elementos generados y así sucesivamente hasta que las iteraciones paren	
5- Resultado de la iteración	Se generan una cantidad indeterminada de ranuras para objetos y pasivas. Estas ranuras están definidas por etiquetas
6- Rellenado de ranuras de objeto	Las ranuras de objeto representan los tipos de objetos que puede llevar el personaje. Se rellenan por el diseñador dependiendo de las etiquetas que tengan asociadas y los objetos que cumplan con esas etiquetas. Pueden quedar vacías lo significa que el personaje no lleva ningún tipo de objeto que cumpla con lo limitado por las etiquetas
7- Equipar los objetos de las ranuras rellenadas	Llenar una ranura con un objeto significa que el personaje tiene ese objeto en su inventario. Sin embargo si se quiere que esté equipado hay que indicarlo expresamente
8- Rellenado de ranuras de pasiva	Las ranuras de pasiva representan los tipos de pasiva que puede albergar el personaje. Si se dejan vacías no llevará pasiva. Si se rellenan con alguna de las que encaje en las etiquetas el personaje la ejecutará como deba durante el juego (en qué momento y a quién)
9- Volcado a base de datos de personaje	Una vez completa la ficha esta se vuelca a la base de datos de personajes

Tabla 4.2.3.1 Diseño de alto nivel del algoritmo de generación de ficha

4.2.3.1 - Base de datos de personajes (singleton)

Cuando la base de datos de personajes sea implementada debería ser solamente una. Contendría a todos los personajes del juego (independientemente

de que participen o no en batallas) con sus estadísticas actualizadas. Usar un patrón **singleton** (una sola instancia de un objeto accesible desde cualquier parte del código en ejecución de TRPGMaker) ayudaría a la comprensión y seguridad.

Además esta clase actuará como controlador a la hora de crear personajes decidiendo cuáles entran en la base de datos y cuáles dependiendo de que cumplan requisitos por determinar (Tabla 4.2.3.1). Sin embargo la actualización de personajes y sus estadísticas será controlada por eventos provinientes del sistema de batalla.

4.2.3.2 - Clase para la ficha de personaje y sus relaciones

Debería existir **una clase para la ficha de personaje**. El singleton a su vez tendría guardados conjuntos de esas fichas (representados en la Figura 4.2.3.2.1 con la flecha acabada en rombo negro, también llamada *composición*).

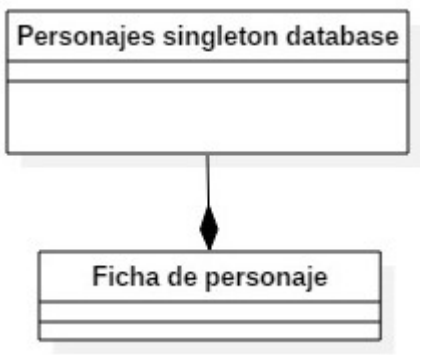


Figura 4.2.3.2.1 Diseño de alto nivel de las clases de la base de datos de personajes (el rombo negro indican composición)

4.2.3.3 - Editor para la ficha de personaje y su relación con la clase

La relación entre el editor de personaje y la clase de la ficha de personaje se aprecia en la figura 4.2.3.3.1. Hay una interdependencia (línea discontinua en ambos sentidos) que identifica que para crear fichas de personaje hace falta un editor de fichas de personaje.

4.2.3.3.1 Diseño de alto nivel de dependencias de la clase de ficha de personaje y su editor (la doble línea discontinua indica interdependencia)

4.2.3.4 - La conexión de los personajes con el sistema de batalla

La información de los personajes se mantiene almacenada siempre, pero los cambios que se producen durante la ejecución del juego no se almacenan en la ficha de personaje original. Estos cambios, como la vida perdida, el maná gastado o la experiencia ganada deberán ser persistidos en la información del estado del juego utilizando alguna de las interfaces proporcionadas por Unity para ello. Además, a excepción de la experiencia, cualquier pérdida de salud o de maná y cambios de estado, generalmente en los juegos tácticos, se resetearán entre cada combate.

PARA IMPLEMENTACIÓN: Para generar un TRPG Character es tan sencillo como añadirle a una decoración existente en el mapa del juego un nuevo Elemento de tipo TRPGCharacter desde Unity y automáticamente se cargarán las opciones disponibles para esa decoración, que de ahora en adelante se convertirá en un Personaje.

4.3 - Sistema de combate táctico y conexión con IsoUnity

4.3.1 - Desarrollo del sistema de combate táctico

El sistema de combate táctico es el tipo de combate que se ha decidido para este proyecto como objetivo principal. Para conseguir un resultado correcto se han estudiado otros juegos que hacen uso de un sistema de combate por turnos para generar un sistema propio que contenga las mejores ideas de otros proyectos.

Por ello el sistema de combate de TRPG está basado en las estadísticas de los personajes jugables existentes en el mapa. Dentro de las organizaciones que hemos analizado en la posible arquitectura del turno, para este desarrollo tomaremos aquel basado en la de velocidad, se ordenan los turnos de cada personaje, siendo el más rápido el primero en poder ejecutar sus órdenes.

Cuando ya se ha decidido el orden de juego, cada personaje dispone de un turno por ronda para poder realizar sus acciones. Al terminar de realizar todas sus posibilidades el personaje dejará de ser controlable hasta que todos los demás hayan terminado sus movimientos y hayan entrado en estado de defensa.

Un turno engloba todas las posibles opciones de las que puede hacer uso un personaje hasta que el personaje pasa al estado defensa. Defensa es el estado final del turno, ya que cierra todos los demás estados y bloquea cualquier acción para ese personaje.

Una ronda engloba los turnos de todos los personajes jugables en el mapa, por ello, hasta que el objetivo se haya completado o hasta que todos los personajes jugables del mapa hayan sido eliminados, las rondas irán sucediendo una tras otra hasta que cualquiera de los dos casos mencionados anteriormente se hayan cumplido.

En el desarrollo de cada Turno, un personaje tiene acceso a cuatro acciones diferentes: atacar, usar habilidad, moverse y defender.

Atacar: Permite al personaje realizar una acción básica a un objetivo utilizando su arma principal. Esta acción resultará normalmente en la realización de daño de ataque básico (según el arma que lleve equipada) sobre el objetivo. Podrían darse excepciones, por ejemplo, en que un arma de fuego realizara daño de fuego que sanara a criaturas afines al fuego. Por otro lado, determinadas armas podrían afectar a otro número de casillas que excedan la casilla adyacente.

Usar habilidad: Se muestra una lista de habilidades disponibles para el personaje actual y si se dispone de suficiente PM (puntos de maná) se podrá llegar a realizar uso de esas habilidades seleccionando al objetivo según el alcance de la habilidad seleccionada.

Ambas acciones son categorizadas como acciones de combate, que sólo pueden ejecutarse una vez por turno. Esta característica fue definida gracias al periodo de pruebas que reveló que no es adecuado que un personaje pueda realizar dos acciones de daño durante su turno, ya que eso podría provocar que el juego sea demasiado sencillo o complicado si el jugador no entiende bien el funcionamiento del turno.

Mover: esta opción permite al personaje moverse por el mapa para poder acercarse a objetivos lejanos a los que no se puede alcanzar con las habilidades o el ataque básico. La cantidad de casillas por las que se puede mover un personaje depende de los atributos que tenga ese personaje en la base de datos. Dado que es un movimiento físico, dicho movimiento se puede bloquear (tanto de aliados como enemigos) ya que los personajes bloquean las rutas disponibles detrás de ellos. Esta acción solo se puede realizar una vez por turno ya que no queremos que existan personajes que puedan moverse por todo el mapa a su antojo. Esta cualidad es, además, una de las características que definen la estrategia a seguir durante un combate dado que caracterizan las opciones defensivas y ofensivas.

Defender: la acción de defender provoca que el personaje pase a un estado de turno finalizado en el que ya no puede hacer más acciones y pasa turno. Por ello defender siempre es la última acción a realizar en el turno de cada personaje.

4.3.2 - Turno

El turno (Turn) es la clase encargada de definir el funcionamiento de cada turno del juego, por ello es una de las clases más importantes y necesarias del juego. Es básico que el desarrollador conozca esta clase para poder tener control sobre la forma que tendrán los combates en su juego, ya que define el funcionamiento de las batallas y los movimientos y funciones que puede realizar un personaje durante su estado de ataque.

El turno se define por las fases que lo forman. El turno en general se define en base a la forma en la que se organiza la actuación de las unidades. Esta organización puede realizarse de diversas formas entre las cuales destacamos: por unidad y por equipo. Dentro de una organización por unidad, las unidades realizan un único turno cada vez independientemente del equipo al que pertenezcan, pudiendo ordenarse por atributos del juego como la velocidad o puntos de acción disponibles. Por el contrario, cuando se organizan por equipo, todas las unidades de un mismo equipo podrán actuar libremente en el orden que desee el jugador hasta que se complete el turno y, posteriormente, éste pasará al siguiente equipo. Dentro de un turno podemos encontrar diferentes fases que representan un tipo de acción, dentro de las cuales definimos el uso de un comando (atacar, usar habilidad o usar un objeto, por ejemplo), moverse o defenderse.

Para que el turno funcione, se debe añadir al Game creado anteriormente en IsoUnity un atributo de la clase Turn, que se encargará de ir otorgando el control de los personajes existentes en el Mapa según las normas especificadas en la propia clase (actualmente el personaje más rápido es el personaje que se mueve primero).

El turno también especifica las posibles transiciones que pueden darse entre el uso de las acciones, es decir, define el ciclo del turno. Esto permite que se limiten el uso de acciones según aquellas que se hayan utilizado. Por ejemplo, si un jugador decide utilizar un objeto no podrá lanzar un hechizo durante el mismo turno. Esta clase es la que define cómo se va a desarrollar cada partida.

Actualmente, el turno es una máquina de estados que permite navegar entre ellos siempre y cuando se vayan realizando las diferentes acciones disponibles. Se dispone de cuatro estados: Atacar, Habilidad, Movimiento y Defender. Se puede acceder a todos los estados desde cualquiera de ellos, pero existen una serie de

restricciones, ya que al usar el estado atacar se cierra la posibilidad de usar una habilidad, o al llegar al estado defender se acaba el turno y se pasa a controlar al siguiente personaje.

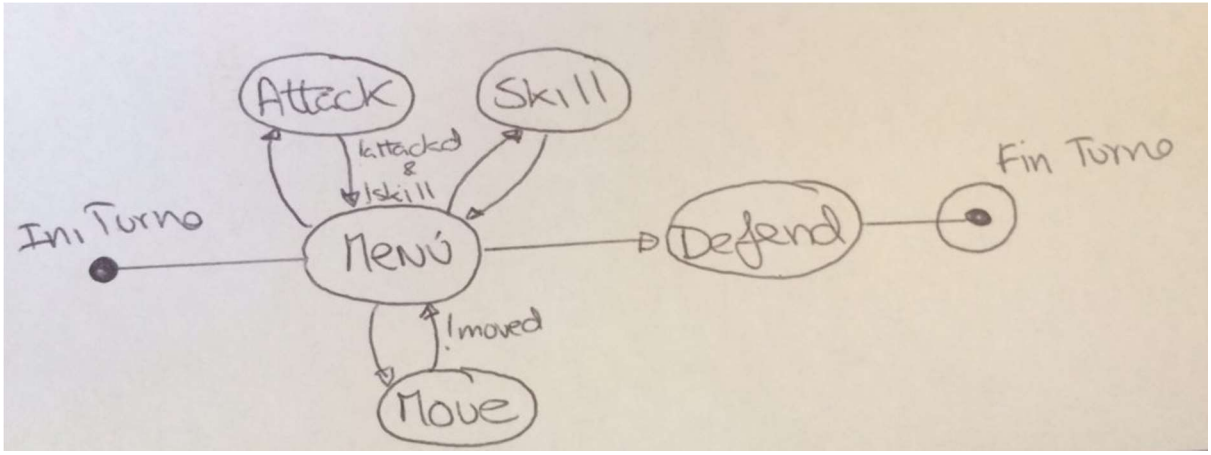


Figura 4.1.3.1 Posibilidades durante un turno

4.3.3 - Conexión con *IsoUnity*

Existen dos grandes conexiones con IsoUnity y una más pequeña, cada conexión cumple su función y conecta un apartado de TRPG con diferentes aspectos del proyecto de los hermanos Pérez Colado.

Mapa: El mapa de IsoUnity está compuesto por muchas celdas con diferente contenido en ellas. Por ello se decidió que todo el sistema de combate y selección de celdas utilizaría un “selector” que permitiría analizar las celdas existentes alrededor del personaje jugable y que marcaría las celdas disponibles para la acción de un color especial, para que el jugador entienda cuales son las casillas seleccionables. Esta clase que mezcla el Mapa de IsoUnity y TRPG se llama **Painter** y es la encargada del estudio del acceso y localización de las celdas del mapa para el uso de las habilidades o acciones del personaje.

Game: El juego de IsoUnity se encarga de unir el mapa creado anteriormente con las herramientas de IsoUnity con el sistema de turnos y los personajes que aparecen en el mapa. Para ello se debe incluir la clase turno como atributo a la clase Game existente en las escena, para que nada más arrancar el juego esta

clase turno haga los análisis necesarios y localice todos los personajes jugables del mapa.

Painter: Es la clase encargada de fusionar el mapa con los jugadores de TRPG, se trata de una clase que partiendo de unos parámetros va seleccionando casillas disponibles para las diferentes acciones posibles de un jugador, aunque no parezca demasiado importante esta clase es la que realiza la conexión entre IsoUnity y TRPG y es la encargada de fusionar estos dos proyectos para que funcionen como uno solo. Esta clase lleva incluida una flecha de selección que será la encargada de indicar el jugador en qué casilla está haciendo clic y cuál va a ser la casilla destino u objetivo de cualquier acción. Recordad que esta clase delimita las acciones a las casillas que estén al alcance por lo que si se selecciona fuera de la casilla no ocurrirá nada.

4.4 - Juego demostrativo

Con el objetivo final en mente se quiere entregar, junto con la memoria y con la información del proyecto, una demostración de lo que hemos obtenido durante el desarrollo del trabajo. Para ellos hablaremos de la temática, funciones y posibilidades que ofrecerá esta demostración.

Temática: La demostración estará basada en el mundo de Juego de Tronos, y dispondrá de varios personajes jugables con diferentes fichas de personaje. Así, se podrá jugar con un equipo formado por un guerrera y un hechicero que se enfrentarán en combate contra dos enemigos armados.

Funciones disponibles: el jugador podrá disfrutar de todo lo mencionado en este proyecto y tendrá total libertad para crear, editar o modificar cualquier atributo de los personajes y/o habilidades. Nuestra intención es ofrecer una demostración abierta para que se puedan revisar todos los aspectos implementados en la herramienta.

Objetivo: El objetivo de la demostración es vencer a los personajes no jugables que están situados en el mapa, se les puede atacar con ataques físicos y/o habilidades. Al terminar con la vida de los enemigos, la partida terminará y se

acabará la demostración. Siempre se puede reiniciar para seguir probando estrategias diferentes.

Guía de uso: En la propia entrega de la demostración incluimos un documento .txt con las instrucciones para probar la demo. Aunque nos hubiese gustado utilizar un archivo ejecutable, lo más sencillo es que se ejecute en el propio entorno de Unity.

Capítulo 5: Implementación, pruebas y resultados

A continuación, se detalla el proceso de implementación, pruebas y resultados que se han conseguido durante todo el proyecto.

5.1 - De los prototipos de editores a los editores finales

El primer paso que se llevó a cabo fue la implementación de una serie de editores en Unity que permitieran al usuario crear elementos para su utilización a lo largo de la ejecución del juego.

Para ello se tomaron como base los editores descritos en la fase de diseño que se basaban a su vez en las especificaciones previamente habían sido dadas en la fase de especificación.

Como también hemos comentado previamente, el modelo de desarrollo ha sido iterativo e incremental utilizando un prototipado ágil. Así hemos ido creciendo desde editores funcionales simples hasta editores cada vez más complejos que se han mostrado adecuados para nuestros propósitos. En los siguientes subapartados explicaremos lo que los editores finales cumplen (en formato regular) y lo que no cumplen (~~en formato tachado~~) respecto a lo que nos habíamos propuesto inicialmente.

5.1.1 - Editor de atributos

Requisitos sobre el editor de atributos
-Dar nombre -Dar id (string) -Indicar si es básico, central o derivado -Dar descripción ~~-Dar avatar~~ -Dar valor predeterminado -BD: nuevo, guardar, cargar, editar, eliminar

El primer editor implementado fue el de atributos porque los atributos son los elementos básicos del juego. Contempla todas las especificaciones indicadas excepto la de dar avatar (se demostró no esencial en las iteraciones).

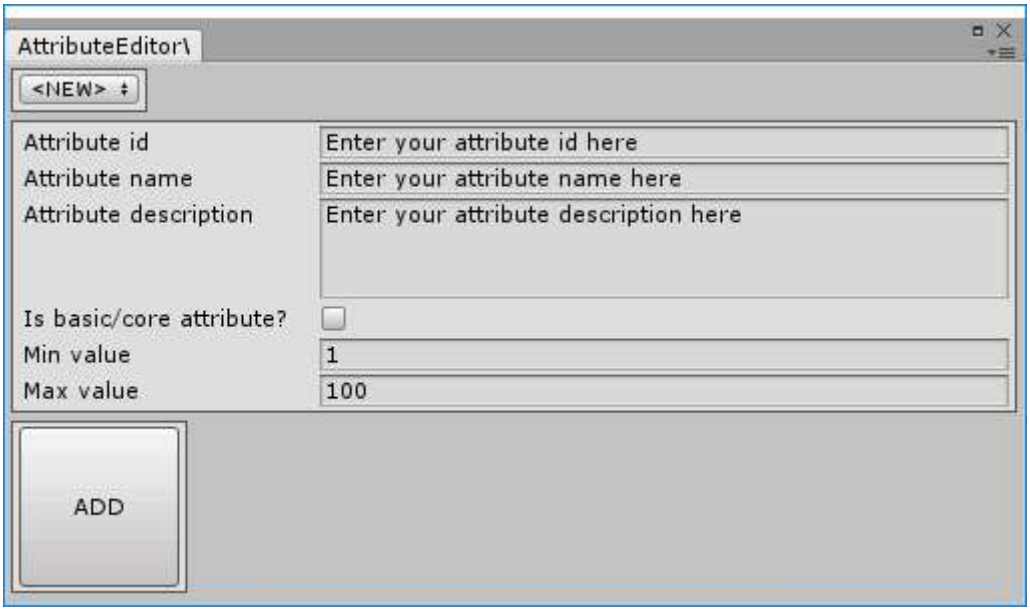


Figura 5.1.1.1 Editor de atributos para nuevo atributo

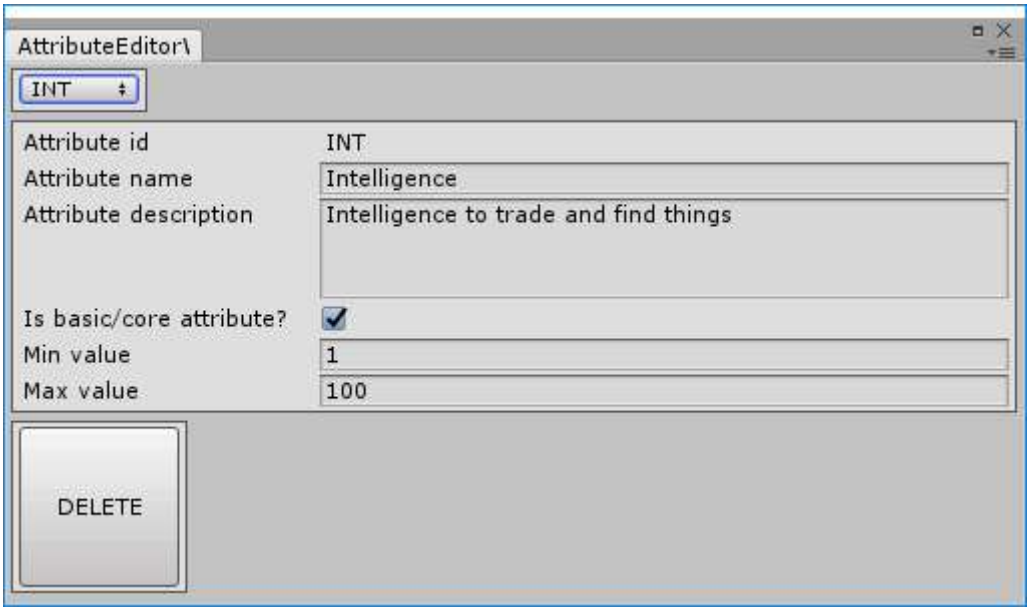


Figura 5.1.1.2 Editor de atributos editando el atributo “Inteligencia”

5.1.2 - Editor de etiquetas

Requisitos sobre el editor de etiquetas
-Dar nombre (actua de id y es autodescriptivo) -BD: nuevo, guardar, cargar, eliminar

Como hemos hablado en la especificación y el diseño las etiquetas son un elemento clave para relacionar todos los elementos del sistema. Un editor simple que pertime añadirlas y eliminarlas del sistema y decir con qué elemento de este se relacionan forma el prototipo final.

Figura 5.1.2.1 Editor de etiquetas añadiendo una etiqueta nueva sobre cuándo se ejecuta una pasiva

5.1.3 - Editor de fórmulas generales

Requisitos sobre el editor de fórmulas
~~-Dar nombre (actúa de id)~~ ~~-Dar descripción~~ ~~-Indicar a qué atributo afecta (mediante su id)~~ ~~-Dar la propia fórmula~~ ~~-Dar la prioridad~~ ~~-BD: nuevo, guardar, cargar, editar, eliminar -BD: nuevo,~~

~~guardar, cargar, eliminar~~

Este editor no se ha llegado a implementar porque las fórmulas de la versión que hemos desarrollado de TRPGMaker son simples valores absolutos que se aplican a objetos, pasivas y clases y la prioridad en la que se ejecutan viene dictaminada directamente por el algoritmo de creación de ficha (explicado en la fase de diseño). Esas fórmulas no requiren de ningún menú ya que son insertadas "inline" cuando se editan objetos, pasivas o clases como veremos en los siguientes menús.

El desarrollo y mejora de las fórmulas y su editor son claramente una parte que queda pendiente para trabajo futuro. Al tener ya bastante avanzados su especificación y diseño y gracias el modelo de desarrollo que hemos seguido hacen que su futura integración no sea traumática.

5.1.4 - Editor de pasivas

Requisitos sobre el editor de etiquetas
-Dar nombre (actúa de id) -Dar descripción ~~-Dar avatar~~ -Dar tags de tipo "cuándo" y de tipo "a quién" -Dar fórmulas ~~y sus prioridades~~ -Permitir slots configurables de tipo pasiva -BD: nuevo, guardar, cargar, editar, eliminar

El editor de pasivas contempla todo lo dicho en la especificación excepto la inclusión de un avatar (mismo caso que editor de atributos) y las prioridades de las fórmulas (tal como hemos explicado en el editor de fórmulas). Podemos llegar a incluír hasta 30 etiquetas sobre "cuándo" y "a quién" se ejecuta la pasiva, 10 fórmulas relacionadas con la ejecución de la pasiva y 24 ranuras que pueden ser rellenadas por otras pasivas.

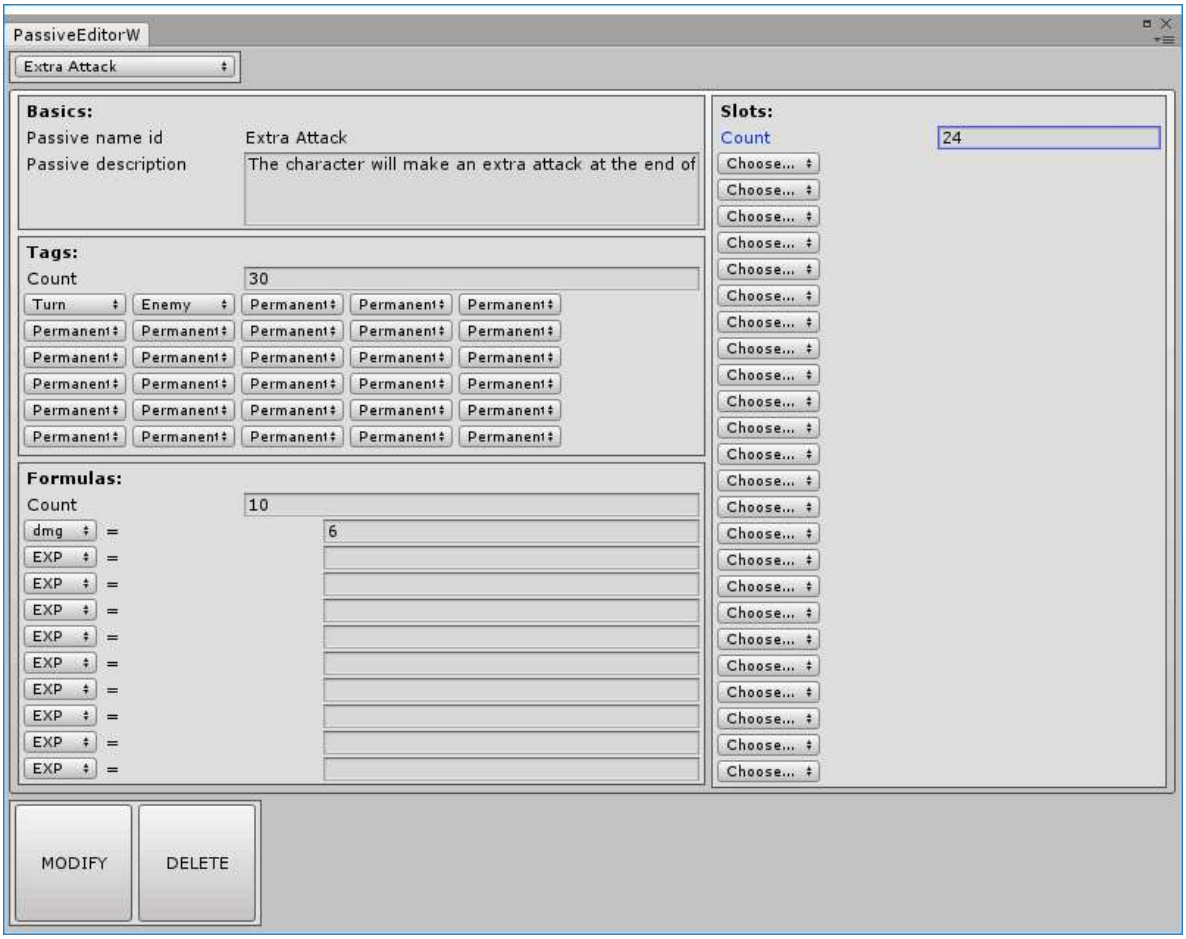


Figura 5.1.4.1 Editor de la pasiva “Extra attack” con dos etiquetas y una fórmula. Además se muestra la posibilidad añadir más fórmulas, etiquetas y ranuras modificando las propiedades “Count” del editor

5.1.5 - Editor de objetos

Requisitos sobre el editor de objetos
-Dar nombre (actúa de id) -Dar descripción ~~-Dar avatar~~ -Dar tags -Dar fórmulas ~~y sus prioridades~~ -Permitir slots configurables de tipo ítem y de tipo pasiva

-BD: nuevo, guardar, cargar, editar, eliminar

El editor de objetos contempla todo lo dicho en la especificación excepto la inclusión de un avatar (mismo caso que editor de atributos o editor de pasivas) y las prioridades de las fórmulas (tal como hemos explicado en el editor de fórmulas). Podemos llegar a incluír hasta 30 etiquetas sobre el objeto a elección del diseñador (en que parte del cuerpo va, si es objeto mágico o no…), 10 fórmulas relacionadas con el objeto (su daño, su defensa…) y 24 ranuras que pueden ser rellenadas por otros objetos (addons al objeto actual) o pasivas.

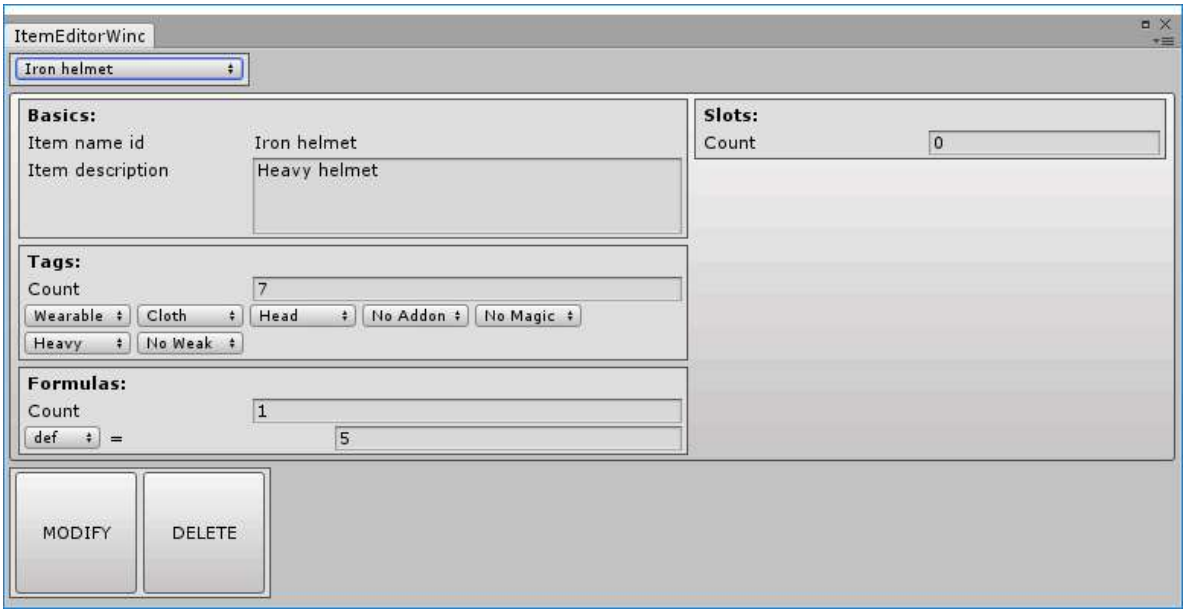


Figura 5.1.5.1 Editor del objeto “Iron helmet” con varias etiquetas que describen el objeto y una fórmula que indica la defensa del objeto. Además se muestra la posibilidad añadir más fórmulas, etiquetas y ranuras modificando las propiedades “Count” del editor

5.1.6 - Editor de clases y especializaciones

Requisitos sobre el editor de clases y especializaciones
-Dar nombre (actúa de id)

-Dar descripción
~~-Dar avatar~~
-Dar fórmulas ~~y sus prioridades~~
-Permitir slots configurables de tipo ítem, de tipo especialización y de tipo pasiva
-BD: nuevo, guardar, cargar, editar, eliminar

El editor de pasivas contempla todo lo dicho en la especificación excepto la inclusión de un avatar (mismo caso que editor de atributos, editor de pasivas o editor de objetivos) y las prioridades de las fórmulas (tal como hemos explicado en el editor de fórmulas).

Las clases y especializaciones no incluyen etiquetas que las describan. Como hemos visto en el diseño hemos decidido que los personajes del juego puedan tener una sola clase básica (en la Figura 5.1.6.1 estamos editando la clase “Human” que es básica y vemos en sus ranuras que tenemos una de tipo “Specialization” en la cual como muestra la Figura 5.1.6.2 podemos especializar a un personaje en las clases que queramos. En este caso “Swordman” y “Bowman”). Podemos rellenar hasta 10 fórmulas relacionadas con la clase (o especialización) que estemos editando y 24 ranuras que pueden ser rellenadas por otros tipos de objetos, pasivas y especializaciones.

Hay que destacar que la jerarquía de clases está limitada a dos niveles (clase y especialización) para el editor final pero que el sistema es suficientemente flexible para poder extender esta jerarquía a más niveles de especialización. En cualquier caso es algo a desarrollar en un futuro.

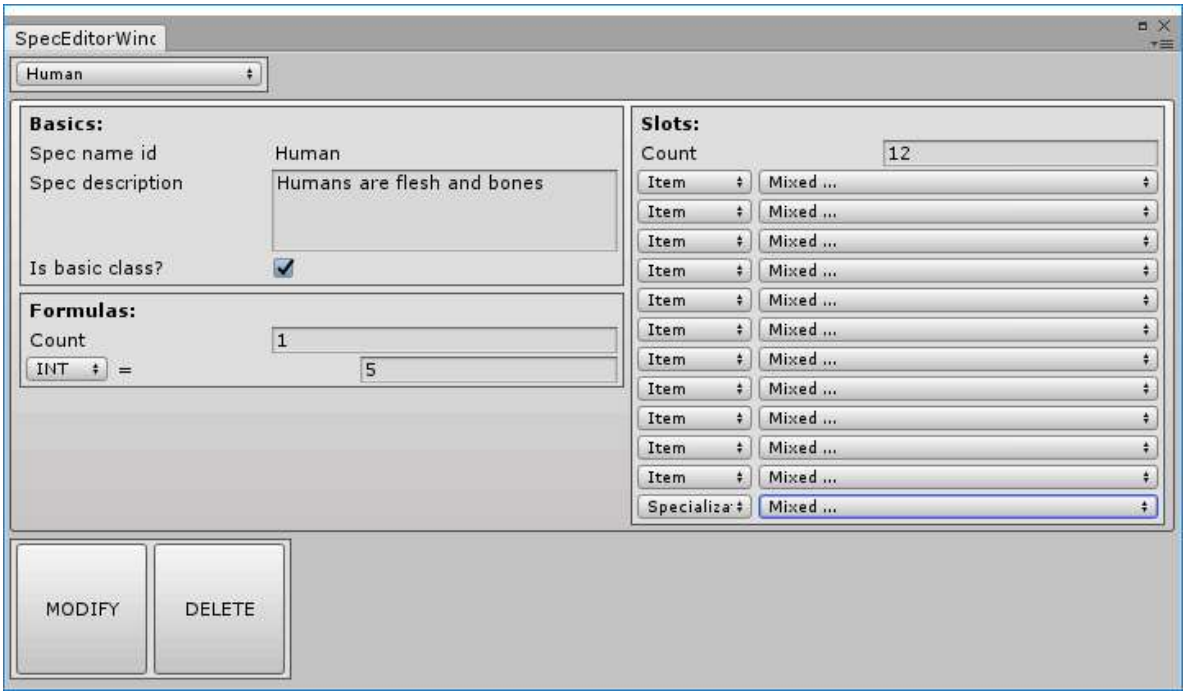


Figura 5.1.6.1 Editor de la clase “Human” con una fórmula y con ranuras para objetos y especialización. Se pueden añadir más fórmulas y ranuras (de tipo objeto, especialización y pasiva).

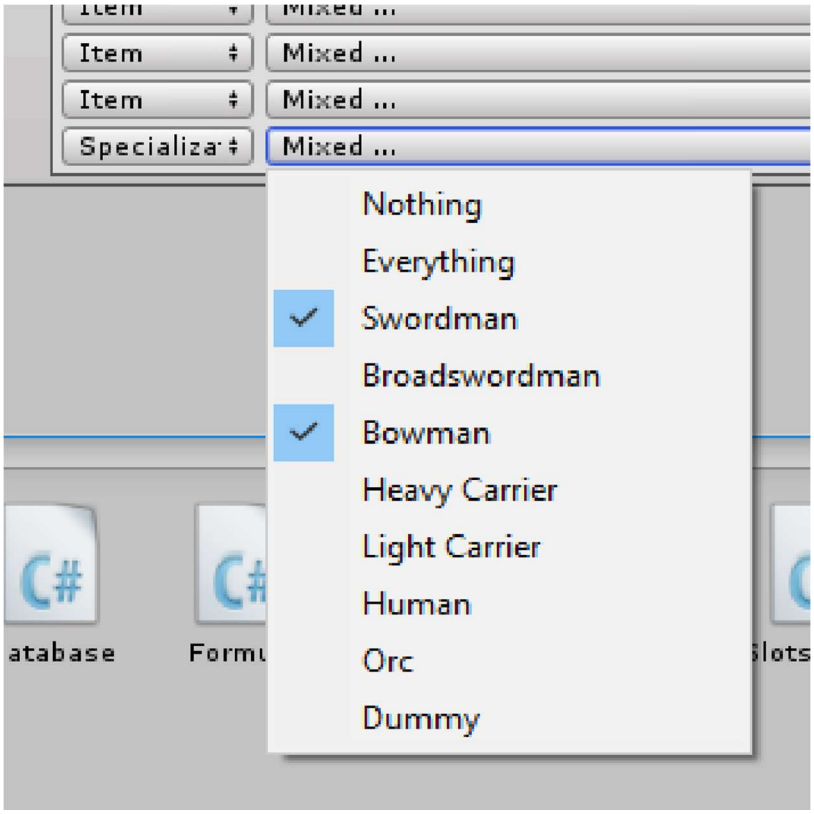


Figura 5.1.6.2 Detalle de la ranura “Specialization” del Editor de la clase “Human”. Vemos que hemos elegido dos posibles especializaciones: “Swordman” y “Bowman”

5.1.7 - Editor de habilidades

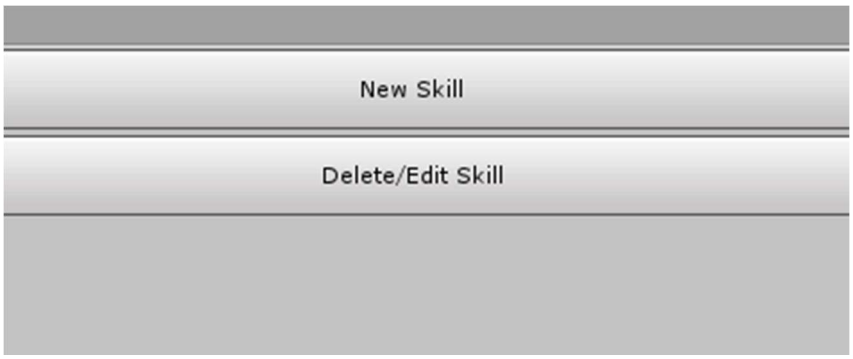


Figura 5.1.1.1 Creación de habilidades

El menú de habilidades, en su primera fase de desarrollo se implementó para dar al usuario la posibilidad de crear una habilidad con un nombre, una descripción y una cantidad de daño, pero pronto se echaron en falta diferentes opciones como el alcance o el tipo de daño que la habilidad iba a provocar. Por ello a medida que se fue ideando el sistema de combate fueron ampliando las diferentes opciones que otorgaba este menú. A continuación se detallan las opciones finales que se han implementado en este editor para que el usuario entienda para que se utiliza cada opción:

El menú consta de dos opciones principales, crear habilidad y eliminar/editar las habilidades ya creadas, se explicarán paso por paso:

1) Creación de una habilidad:

   Al entrar el en menú de creación se dispone de unos campos de texto para rellenar la información de la habilidad y de una serie de parámetros que definen el funcionamiento de la habilidad.

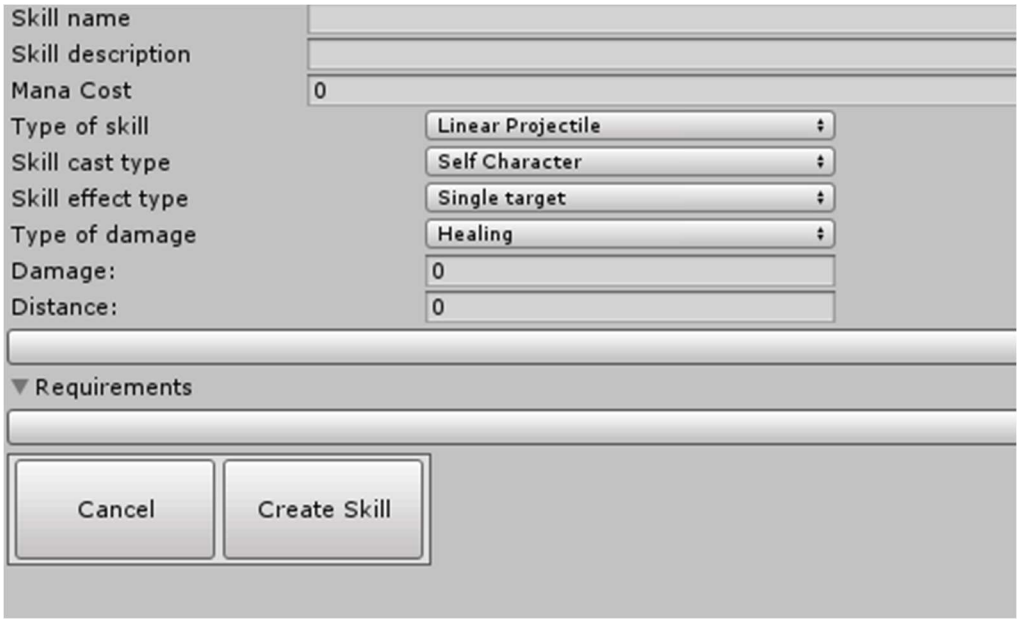


Figura 5.1.1.2 Creación de una habilidad

Los tres primeros campos permiten rellenar información básica de la habilidad, añadiendo un nombre, una descripción y un coste (un entero) de puntos de maná para usar la habilidad.

El tipo de habilidad permite definir cómo será el movimiento de la habilidad a lo largo del mapa, pudiendo ser por ejemplo un movimiento en línea recta, una parábola o un efecto global.
El tipo de "casteo" de habilidad define desde dónde se lanza la habilidad, pudiendo ser desde el propio lanzador, desde una celda elegible dentro del posible rango de la habilidad o desde un punto en el cielo que tendrá alcance global.
El tipo de efecto de la habilidad define a quién afecta la habilidad, dando la posibilidad de que las habilidades funcionen en un área pequeña, una sola casilla o por el contrario, el efecto repercuta en una serie de casillas alrededor de la celda que el jugador seleccione.
El tipo de daño describe si la habilidad hace daño, sanación o ambos.

Por último quedan dos campos de enteros a rellenar que definen el daño y la distancia de la habilidad, ajustando así su alcance y la cantidad de daño que provoca la habilidad creada.

Para terminar el proceso de creación de habilidad se pulsa el botón "Create Skill" que almacena la habilidad en un asset que permite la carga de las habilidades creadas al iniciar el proyecto.

2) Editar/Eliminar una habilidad ya creada:

El menú de administración de habilidades permite acceder al listado de habilidades y, para cada una, poder editarla o eliminarla. La figura 5.1.1.3 muestra una serie de habilidades ya creadas y su posible edición. Desde este panel, al contrario que al crear, no es posible modificar todos los campos de una habilidad ya que se utilizan como claves en otros lugares. Por ello si al editar alguna habilidad no se consigue el resultado esperado, la opción que se ofrece es eliminarla y volver a crearla.

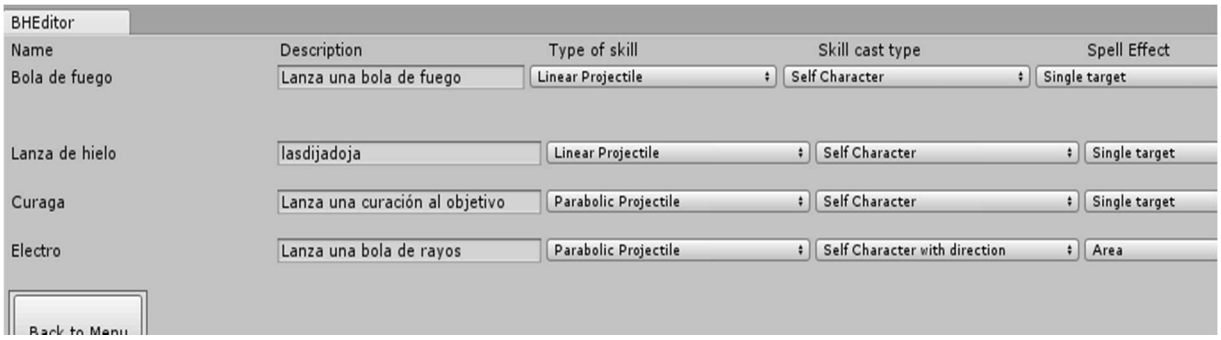


Figura 5.1.1.3 Edición de habilidades

5.1.8 - Editor de personajes

Requisitos sobre el editor de personajes
-Dar nombre (actúa de id) -Dar descripción ~~-Dar avatar~~ -Dar valor de clase y de especialización(es) -Dar valores a los atributos básicos y centrales -Generación de slots de ítems y pasivas -Dar valor a los slots de ítems y pasivas -Decidir qué ítems de los puestos en los slots van equipados -BD: nuevo, guardar, ~~cargar, editar, eliminar~~

El editor de personajes ha sido el más complicado de materializar por los diversos diseños que se plantearon para la creación de la ficha del personaje tal como ya hemos comentado en la sección “4.2.3 Base de datos de personajes, clases y editor” por tanto es el que hemos elegido para explicar en mayor profundidad. En la sección “5.2.1 Implementación del algoritmo de creación de ficha del personaje” comentaremos también cómo se genera la ficha de personaje a partir del editor de personajes que presentamos aquí.

El editor permite dar nombre (actúa como id) al personaje, añadirle una descripción, fijar sus atributos básicos y centrales, ajustar su clase y especialización, rellenar sus ranuras de objetos y pasivas según la clase y especialización elegidas,

e indicar cuáles de esos objetos de las ranuras van equipados realmente en el personaje.

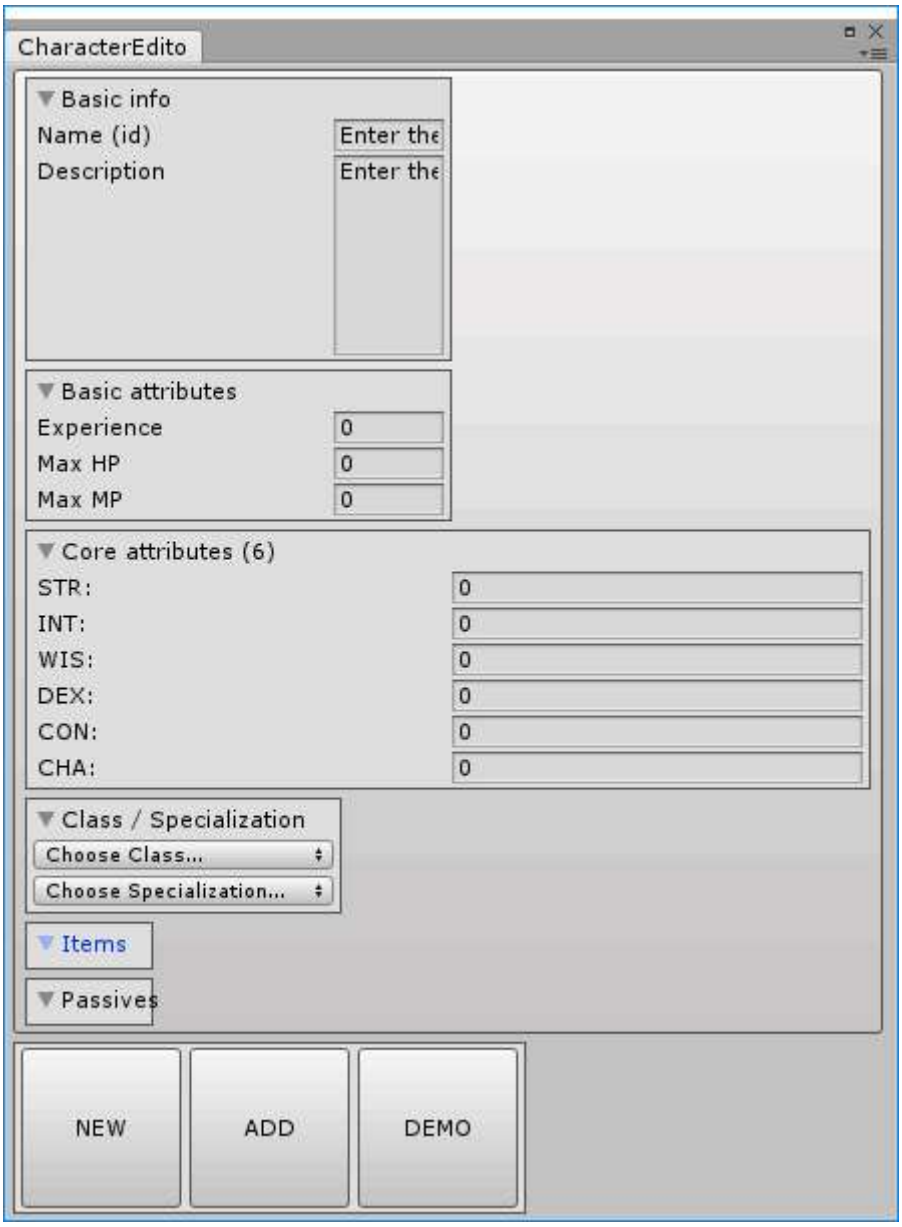


Figura 5.1.8.1 El editor de personajes sin rellenar

La figura 5.1.8.1 muestra el editor de personajes cuando se abre: la posibilidad de rellenar la información básica (nombre y descripción), los atributos básicos que son comunes a todos los juegos TRPG (experiencia de personaje, HP y MP), los atributos centrales que el diseñador define para cada juego mediante el editor de atributos (en este caso se han elegido los típicos de la saga Dungeons & Dragons comentada en la sección “2. Revisión del estado de la cuestión”), los selectores de clase y especialización, y las ranuras por ahora inexistentes que se generarán para los objetos y las pasivas una vez la clase y especialización se concreten.

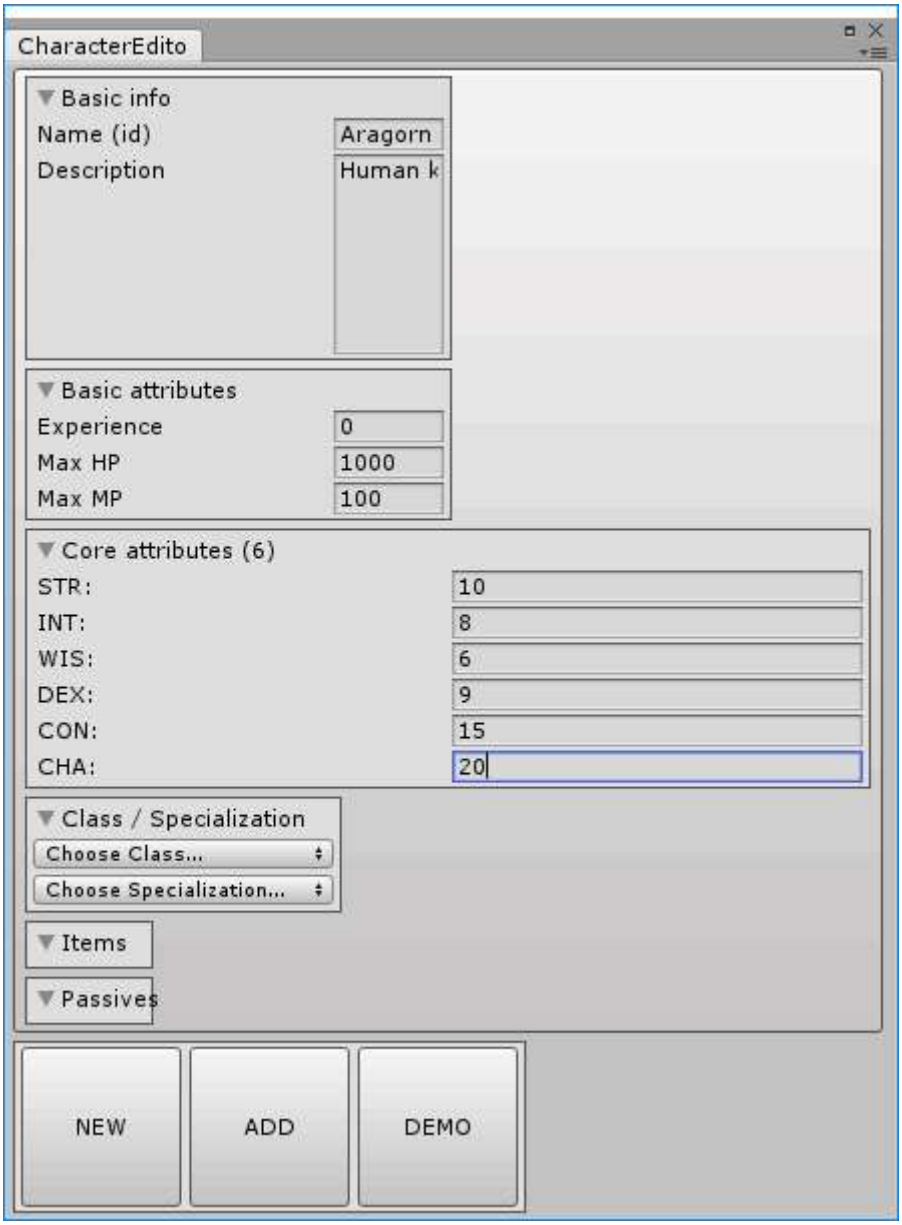


Figura 5.1.8.2 El editor de personajes con información básica y atributos rellenados

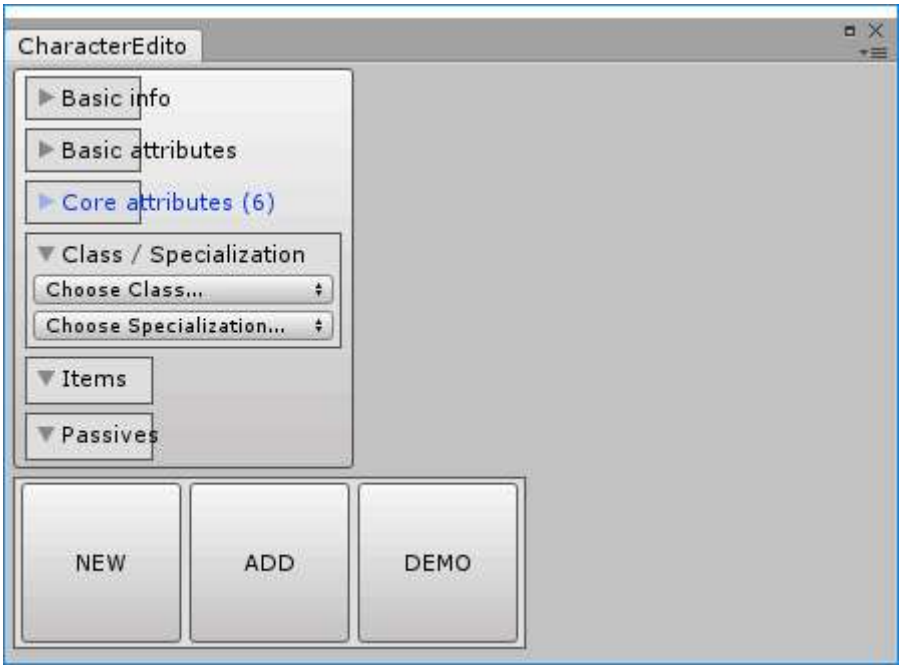


Figura 5.1.8.3 El editor de personajes con información básica y atributos rellenados pero con los apartados colapsados

Una vez rellenados esos apartados (figura 5.1.8.2) tenemos la opción de esconderlos para que no nos estorben usando los *dropdown* que proporciona Unity para sus editores (figura 5.1.8.3). Una vez hecho eso es el momento de fijarse en el selector de clase y especialización.

A modo ilustrativo del proceso iterativo e incremental con prototipos diremos que la primera iteración del editor de personajes consistió en personajes con información básica y atributos básicos. Una segunda iteración incluyó además los atributos centrales.

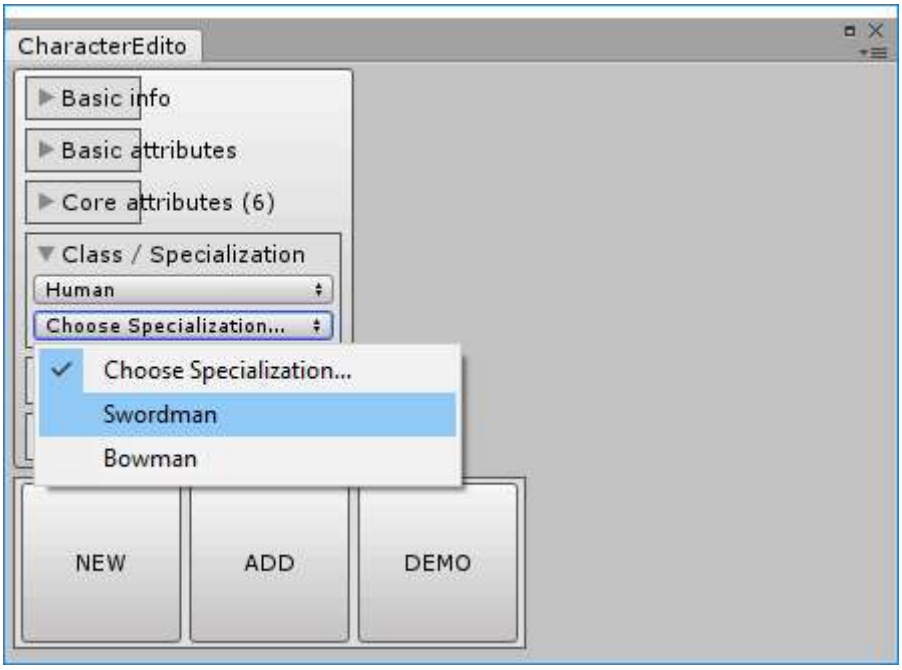


Figura 5.1.8.4 El editor de personajes tras seleccionar la clase “Human” y a punto de seleccionar la especialización “Swordman”

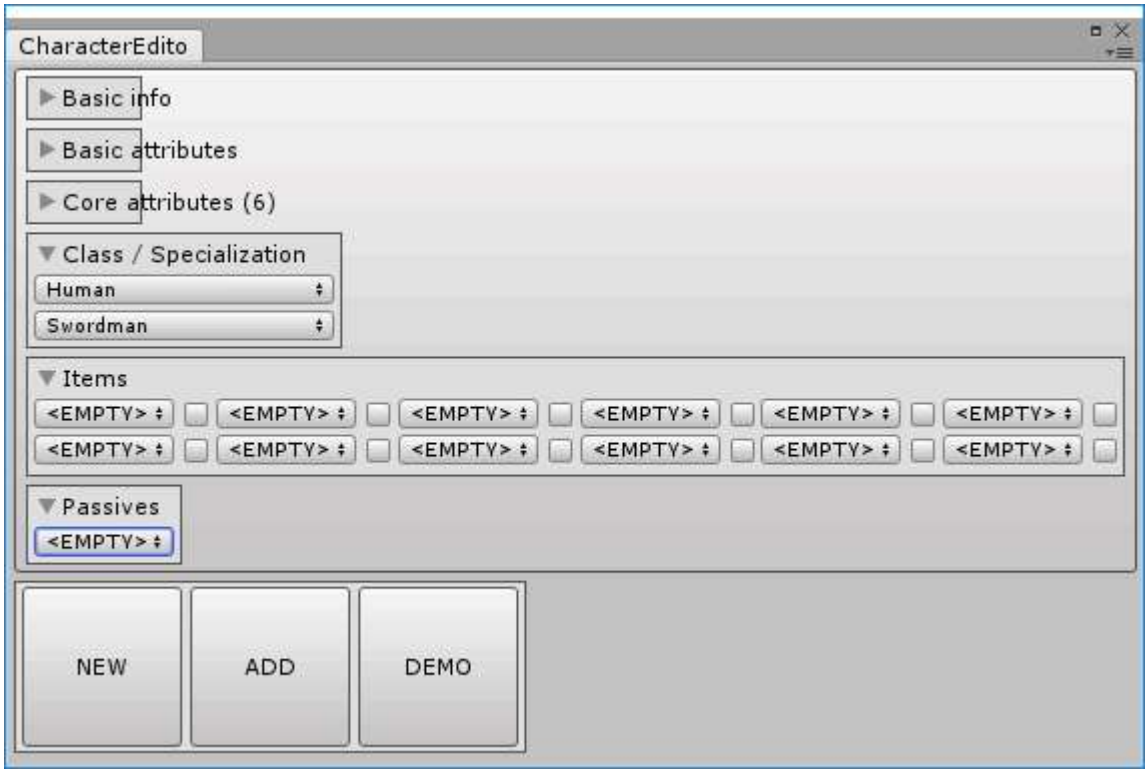


Figura 5.1.8.5 El editor de personajes tras seleccionar la clase “Human” y la especialización “Swordman”

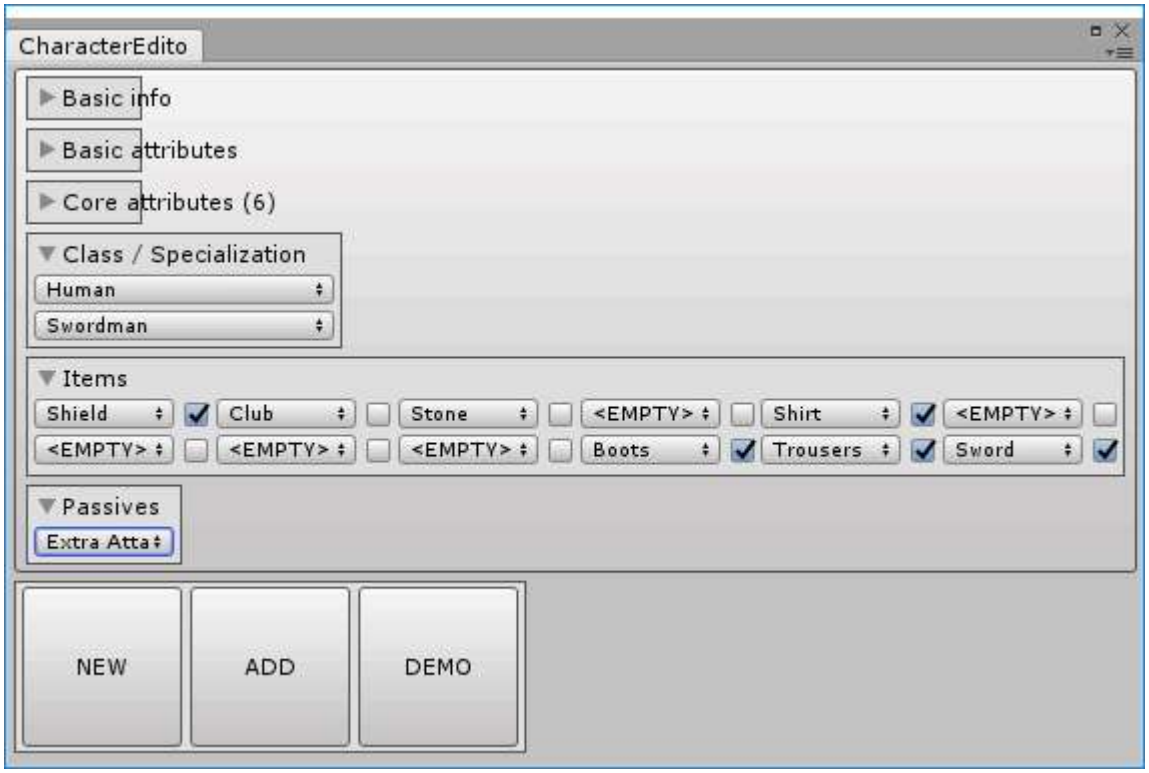


Figura 5.1.8.6 El editor de personajes tras rellenar las ranuras de objetos y pasivas

Cuando concretamos la clase podemos hacer lo propio con la especialización y entonces se generan las ranuras de objetos y pasivas (figuras 5.1.8.4 y 5.1.8.5). Todas estas ranuras nos darán la opción de poderlas rellenar con los objetos y pasivas filtradas según las configuraciones hechas en el resto de editores. Además seleccionando las checkboxes de los objetos podemos indicar que el personaje los llevará equipados y no seleccionándolas que los llevará en su inventario (figura 5.1.8.6). Una vez completado este proceso sólo quedará añadir el personaje a la base de datos de personajes (botón “ADD”).

Cabe destacar que no se han implementado las funcionalidades de cargar, editar o eliminar un personaje de la base de datos. Será algo necesario para el desarrollo futuro de TRPG Maker. Tampoco ha quedado perfecta la maquetación de los menús, siendo este un tema menor (aunque no por ello trivial ni fácil) que también se debería afrontar en el futuro.

El botón “DEMO” que aparece en las capturas simplemente añade varios personajes sin necesidad de rellenar todo a mano. Su finalidad es ilustrar el funcionamiento del menú, generando personajes para una escena de batalla de forma más rápida.

En cuanto al desarrollo iterativo e incremental: el tercer prototipo fue simplemente el que permitía añadir clase y especialización. El cuarto y quinto los que permitían respectivamente ranuras de objetos y ranuras de pasivas. El sexto aunó lo del cuarto y quinto y se quedó como editor final.

5.2 - Implementación de los algoritmos

Dentro de los hitos a conseguir en el desarrollo de TRPG Maker (apartado “3.2.1 Estimación del esfuerzo”) habíamos hablado de los dos hitos más importantes: la creación y gestión de las bases de datos del juego y el sistema de combate.

En primer lugar, **la creación y gestión de las bases de datos del juego** se culmina con la creación de la base de datos de personajes mediante el editor explicado en el apartado “5.1.8 Editor de personajes”. El desarrollo de este editor ha precisado de un algoritmo de creación de ficha de personaje que explicaremos detalladamente en la sección “5.2.1 Implementación del algoritmo de creación de ficha de personaje”.

En segundo lugar, **el sistema de combate** necesita definir el turno y el algoritmo de gestión de turnos. Lo explicaremos en la sección “5.2.2 Turno y algoritmo de gestión de turnos”.

5.2.1 - Implementación del algoritmo de creación de ficha de personaje

Los personajes están representados por la clase *CharacterSheet*. Esta clase tiene los siguientes atributos y estructuras de datos:

Atributos y estructuras de datos de la clase *CharacterSheet*

```
string _nameId;
string _description;
Dictionary<string,double> _baseAttributes;
Dictionary<string,double> _realAttributes;
SpecTemplate _class;
SpecTemplate _specialization;
List<ItemTemplate> _items;
List<PassiveTemplate> _passives;
List<Formula> _tier1Formulas;
List<Formula> _tier2Formulas;
List<Formula> _tier3Formulas;
```

A efectos prácticos una *CharacterSheet* se construye en varios pasos:

1- Constructor inicial
2- Colección de fórmulas
3- Actualización final de los atributos

En primer lugar el **constructor inicial** recibe nombre (`_nameId`), descripción (`_description`), el valor de los atributos básicos y centrales (`_baseAttributes`), clase (`_class`), especialización (`_specialization`), lista de objetos equipados (`_items`) y lista de pasivas (`_passives`) a través del editor de personajes. Es importante resaltar que `_baseAttributes` se construye también incluyendo el resto de atributos existentes en la base de datos general pero con valor 0 (en este momento no afectan al personaje).

En segundo lugar la **colección de fórmulas**: dependiendo de la clase, especialización y lista de objetos equipados vamos a tener distintas fórmulas provenientes de todos ellos. Las volcaremos en los distintos contenedores de fórmulas `_tier1Formulas`, `_tier2Formulas`, y `_tier3Formulas`. Hay que las fórmulas de las pasivas no se tratan en esta versión de la herramienta para los cálculos. Son una de las partes que queda para trabajo futuro.

Las `_tier1Formulas` son fórmulas que modifican los valores de un mismo atributo a través de valores absolutos (ejemplo: defense = defense + 3) y son las que en realidad soporta esta versión de la herramienta.

Las `_tier2Formulas` son fórmulas que modifican los valores de un mismo atributo a través de porcentajes (ejemplo: defense = defense * 0,3). No han sido implementadas, pero se dejan preparados sus contenedores para su implementación futura.

Las `_tier3Formulas` son fórmulas que modifican los valores de otros atributos a través de porcentajes (ejemplo: damage = strength * 0,1). Tampoco han sido implementadas y quedan pendientes de hacerlo en un trabajo futuro.

La idea de tener varios niveles de fórmulas es para que se apliquen en un orden fijo. No es lo mismo aplicar unas fórmulas concretas de nivel 1, 2 y 3 que aplicar esas mismas fórmulas pero en orden inverso. El resultado sería distinto creando entonces juegos inconsistentes.

Y en tercer y último lugar la **actualización final de los atributos**: hay que comprobar que antes de usar las fórmulas almacenadas no se producen cambios en los atributos por los actos ocurridos en la batalla del juego (ataques, defensas, habilidades…). Si ocurren, estos se reciben y actualizan los `_baseAttributes` de manera correspondiente antes de usar las fórmulas almacenadas (que recordemos que en esta versión de TRPGMaker provienen de clases, especializaciones y objetos) en `_tier1Formulas`. Finalmente obtenemos los `_realAttributes` gracias a la base de los `_baseAttributes` y el cálculo de las fórmulas de `_tier1Formulas` teniendo finalmente una ficha actualizada de personaje.

5.2.1.1 - La serialización de la base de datos de personajes

Una parte bastante complicada que no fue contemplada debidamente en la parte de diseño fue el volcado de las bases de datos en archivos persistentes ya que se consideró que su formato no sería dependiente de Unity.

Una vez decidido que eso no sería así y que las bases de datos debían ser assets de Unity, el volcado de la información debía pasar por una fase de serialización. En este caso concreto analizamos la serialización de la base de datos de personajes.

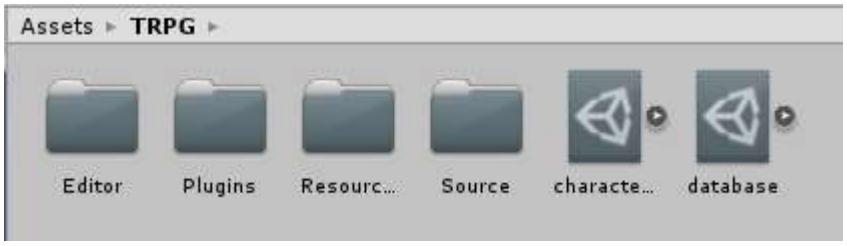


Figura 5.2.1.1 Assets de la base de datos de personajes (characters) y la general (database) tras ser serializados

Serialización de la clase *CharacterSheet*

```
[SerializeField]
string _nameId;
[SerializeField]
string _description;
// Attributes (basic, core, derived)
Dictionary<string,double> _baseAttributes;
[SerializeField]
List<string> _keyBaseAttr;
[SerializeField]
List<double> _valueBaseAttr;
Dictionary<string,double> _realAttributes;
// Class, specialization, items and passives
[SerializeField]
SpecTemplate _class;
[SerializeField]
SpecTemplate _specialization;
[SerializeField]
List<ItemTemplate> _items;
[SerializeField]
List<PassiveTemplate> _passives;
// Collected formulas
List<Formula> _tier1Formulas;
List<Formula> _tier2Formulas;
```

```
List<Formula> _tier3Formulas;
```

Figura 5.2.1.2 Serialización de la clase CharacterSheet

Para serializar atributos es necesario que sean tipos simples e indicarlos con la etiqueta `[SerializeField]`.

Si son estructuras de datos que contienen datos simples sólo se permite serializar listas (`List<>`). Estructuras como `Dictionary<string,double>` deben ser transformadas antes de la serialización en listas como `List<string> _keyBaseAttr` y `List<double> _valueBaseAttr`, todo en un método especial proporcionado por Unity llamado `void OnBeforeSerialize()`. Esto implica que cuando se actúa en el otro sentido (una carga de elementos serializados en la lógica del juego) se deben reconstruir las estructuras de datos originales en otro método especial llamado `void OnAfterDeserialize()`.

El propio volcado a un asset se consigue con los métodos de Unity `CreateAsset()`, `AddObjectToAsset()` y `SaveAssets()`.

5.2.2 - Turno y algoritmo de gestión de turnos

En un principio los personajes solo podían moverse y atacar, pero provocaba que hubiese poca acción en el combate. Por ello se decidió añadir las opciones de “usar habilidad” y “defender”.a las posibilidades de acción de cada personaje.

Por último, aunque no está implementado, se planea otorgar a las habilidades opciones para que modifiquen la velocidad del objetivo, de tal manera que afecten a la velocidad de los personajes y estos puedan llegar a atacar más de una vez en un mismo turno o que ataquen antes.

5.3 - Pruebas y funcionamiento

A medida que se han ido implementando las herramientas se han ido probando y comprobando su funcionamiento. Para ello, en un principio solo se comprobaba si los cambios, las creaciones de nuevos objetos y los diferentes atributos se iban almacenando y actualizando en la base de datos. Cuando se pudo crear un entorno completo se pudo realizar el periodo de pruebas y funcionamiento de cada herramienta.

Para este apartado se ha creado un mapa y unos personajes que pueden pelear entre ellos para comprobar que todo lo creado anteriormente (equipo, personajes, habilidades, atributos…) afecta en tiempo real al funcionamiento del juego y a su jugabilidad. Para ello se explicarán los cambios y mejoras que se fueron añadiendo a las herramientas a medida que estas se probaban.

5.3.1 - Prueba de Habilidades

En un principio, se creó el editor de habilidades con una base de datos propia que almacenaba los datos básicos que utilizaría el motor del juego. Para ello solo se almacenaban el daño de la habilidad, el nombre y una breve descripción de los que esta hacía al usarse. Poco después el equipo se dio cuenta de que necesitaban más detalles en las habilidades para poder diferenciar diferentes tipos de acción a la hora de elegir la mejor manera de usar cada personaje.

Por ello, en la segunda iteración que se le dio a esta herramienta se añadieron diferentes opciones, como la posibilidad de elegir si una habilidad sana o provoca daño. También se decidió que las habilidades se deberían poder lanzar a distancia, por lo que se añadieron nuevos atributos que permiten especificar a qué distancia son capaces las habilidades de seleccionar a un objetivo.

Con estas opciones implementadas cambió la manera de ver el lanzamiento de habilidades y por ello se añadieron opciones para especificar el tipo de daño que provoca una habilidad (por ejemplo, la habilidad, que ya se puede lanzar a distancia, provoca daño en área al objetivo y a sus celdas contiguas, o si por el contrario solo selecciona a la propia casilla). Además se añadió un opción que permite dibujar el efecto de la habilidad, permitiendo así al desarrollador que una habilidad haga un efecto en línea recta o una “X” en torno al objetivo.

Cuando ya se obtuvo un rendimiento completo y ajustado del sistema de habilidades se llevaron a cabo las pruebas en el entorno. Se crearon unas habilidades básicas que existen desde el principio de la partida y se probaron, pero surgió un problema. Todos los personajes podían hacer uso de cualquier habilidad existente en la base de datos.

Para solucionar el problema de uso, se implementó un sistema de Requisitos que obligan al personaje que va a hacer uso de esa habilidad a cumplir una serie de objetivos antes de poder usarla. Para hacerse una idea, una habilidad llamada “Bola de fuego” se podría limitar a los personajes que pertenezcan a la especialidad de “Mago” y que tengan un nivel mínimo de personaje 10.

Por último, cuando ya se realizaron casi todas las pruebas se decidió añadir un nuevo atributo a las habilidades, que permitiría al motor del juego saber qué tipo de animación se va a realizar. Para explicarlo necesitamos un ejemplo:

Si un personaje utiliza una habilidad, esta puede “lanzarse” desde diferentes localizaciones, puede ser que la habilidad se lance desde el propio personaje (“Lanza de fuego” por ejemplo, lanza una línea de fuego a las 3 casillas que tenga delante el personaje) por lo que es el propio personaje (su celda) desde la que se localiza la habilidad, pero se puede dar el caso en el que el personaje lanza un meteorito al enemigo y por ello, no es el propio personaje el inicio de la habilidad sino un punto en el cielo, por lo que se evitan obstáculos del terreno.

Para solucionar este problema se implementaron los atributos “Tipo de habilidad” y “Tipo de lanzamiento” que permiten seleccionar diferentes opciones a la hora de elegir cómo se moverá y cómo se lanzará la habilidad. Según estos atributos el motor del juego calculará la ruta y trayectoria de la habilidad hacia el objetivo.

Como última mejora a esta herramienta se decidió que las habilidades no deberían ser infinitas, no se deben usar durante todos los turnos, por lo que cada habilidad se puede limitar con el “Coste de Maná” que provoca que el uso de una habilidad elimine puntos de maná al personaje. Así se evita que se haga uso constante de una habilidad que es muy poderosa.

Capítulo 6: Conclusiones

Al comienzo del proyecto el objetivo era claro: realizar una herramienta para el motor gráfico Unity que permitiese crear videojuegos del género estratégico. Este objetivo fue creciendo y ampliándose a medida que los miembros fueron cogiendo experiencia y obteniendo ideas de otras herramientas y juegos ya comentados en capítulos anteriores.

Gracias al apoyo del director y el co-director del proyecto poco a poco se fueron cerrando los objetivos y metas del proyecto y se establecieron unas metas a conseguir a medida que se iba avanzando en la herramienta.

Desde que comenzó este proyecto la visión que nos guiaba era tener una herramienta basada en Unity que permitiera de manera sencilla crear videojuegos de rol táctico, con un sistema de movimiento y combate por turnos y basado en casillas similar al de títulos muy conocidos. Después de todo el trabajo realizado, se puede afirmar que el prototipo funcional que tenemos cumple con el objetivo principal del proyecto.

Actualmente sigue siendo complejo producir un juego de este género, pero aun así confiamos en que con la herramienta TRPG Maker los desarrolladores se animen a traer de vuelta este género a los jugadores de hoy en día. Toda herramienta de este estilo, en una primera versión, resulta compleja, pero hemos realizado esfuerzo en la fase de diseño para tratar de que resulte comprensible y que se pueda trabajar con ella de manera inmediata.

Utilizando la tecnología preexistente de IsoUnity y apoyándose en la filosofía y las bases sentadas por este proyecto, los desarrolladores pueden llegar a crear un juego de rol completo, con sus distintos escenarios, personajes, armas y equipamiento. Todo está desarrollado para darle la máxima libertad al desarrollador de implementar los elementos más usados y más característicos cuando quiera y donde quiera.

Como ya hemos mencionado antes, ahora no solo se pueden crear escenarios isométricos con IsoUnity, sino que además a cada escenario se le pueden añadir complejos sistemas de personajes, habilidades y un control de turnos que le dan a esta herramienta no solo una mejora, sino un alcance mucho mayor que el original.

Los editores gráficos son la parte más importante del proyecto ya que detrás de ellos hay largas horas de trabajo para que todo lo que se edita y se crea en ellos funcione perfectamente y de manera integrada. Todo lo creado en los editores se vuelve persistente y es almacenado para que siempre se disponga de ello al abrir de nuevo el proyecto.

Cada editor tiene sus propias opciones ya explicadas en este trabajo, todos ellos tienen una funcionalidad específica y cuando se termina el trabajo con ellos el turno y el sistema de combate son los encargados de realizar que lo definido en las bases de datos se traduzca en la jugabilidad. La creación de personajes, las diferentes opciones de las habilidades, el equipamiento, las estadísticas o las diferentes ranuras permiten que en cada turno el juego sea distinto, cada personaje único, y cada combate una situación táctica distinta sobre la que planificar hasta el más mínimo movimiento.

Además de los editores se ha añadido un sistema de control de turnos que permite al programador modificar parte del código, directamente sin uso de ningún editor. De esta manera el programador modifica a su gusto cualquier comportamiento que no sea de su agrado en el algoritmo de planificación propuesto. El turno es el motor central de los combates y aunque actualmente consideramos que es bastante completo y genérico, la filosofía del código libre y abierto permite mejorar y añadir nuevas opciones a la herramienta bajo demanda, por lo que hemos dejado abierta la posibilidad de mejorar e implementar nuevas funcionalidades si se necesitan. A medida que pase el tiempo, se podrá ir mejorando y añadiendo más opciones a todas las secciones que hemos explicado antes.

En resumen podemos concluir que TRPG Maker convierte a Unity en una herramienta funcional y muy útil para crear videojuegos de rol táctico, totalmente abierta a modificaciones y mejoras por parte de la comunidad de desarrolladores.

Estamos satisfechos de haber podido contribuir al gran proyecto iniciado por los hermanos Pérez Colado, IsoUnity, y de poner nuestro granito de arena al servicio de la comunidad internacional de desarrolladores independientes de videojuegos.

Conclusions

At the beginning of the project the objective was clear: to make a tool for the graphic engine Unity that allowed to create videogames of the strategic sort. This goal was expanded and expanded as members gained experience and gained insights from other tools and games discussed in earlier chapters. Thanks to the support of the director and the co-director of the project little by little they were closing the objectives and goals of the project and set some goals to achieve as the tool progressed.

Since the beginning of this project the vision that guided us was to have a tool based on Unity that would allow in a simple way to create video games of tactical role, with a system of movement and combat by turns and based on boxes similar to well-known titles.

After all the work done, we can say that the functional prototype we have meets the main objective of the project. Nowadays it is still complex to produce a game of this genre, but we are still confident that with the TRPG Maker tool the developers will be encouraged to bring this genre back to the players of today. Every tool of this style, in a first version, is complex, but we have made an effort in the design phase to try to make it understandable and to work with it immediately. Using IsoUnity's pre-existing technology and building on the philosophy and foundation set by this project, developers can create a complete role-playing game with its different scenarios, characters, weapons and equipment. Everything is developed to give the maximum freedom to the developer to implement the elements most used and most characteristic whenever and wherever.

As already mentioned, nowadays not only can you create isometric scenarios with IsoUnity, but in addition to each scenario you can add complex character systems, skills and turn control that give this tool not only an improvement, but also a far greater scope than the original. Graphic editors are the most important part of the project since behind them there are long hours of work so that everything that is edited and created in them works perfectly and in an integrated way.

Everything created in the editors becomes persistent and is stored so that it is always available when you open the project again. Each editor has its own options already explained in this work, they all have a specific functionality and when the work is finished with them the turn and the combat system are in charge of realizing that what is defined in the databases is translated into the gameplay. The creation of characters, different options of skills, equipment, statistics or different slots allow each game to be different, each unique character, and each combat a different tactical situation on which to plan to the smallest

movement. In addition to the editors has been added a system of control of shifts that allows the programmer to modify part of the code, directly without use of any publisher. In this way the programmer modifies at will any behavior that is not to his liking in the proposed scheduling algorithm.

The turn is the central engine of the fighting and although we now consider it quite complete and generic, the philosophy of free and open code allows to improve and add new options to the tool on demand, so we have left open the possibility of improving and implement new features if needed. As time goes by, you can improve and add more options to all sections that we have explained before.

Contribución

1. Javier Druet Honrubia

Aunque la primera idea que se diseñó no fuese la opción final, participé en el desarrollo del concepto del proyecto. Aunque los dos miembros del trabajo hemos aportado, en mayor o menor cantidad, nuestro granito de arena a cada sección del proyecto, cada uno ha trabajado centrándose en diferentes apartados.

A continuación se listan los apartados que he desarrollado e implementado:

1.1 Creación, gestión, edición y uso de habilidades.
1.2 Creación del sistema de selección para IsoUnity.
1.3 Implementación del sistema de Turnos, acciones y movimientos.
1.4 Unión de los editores para su uso durante el combate.
1.5 Creación del escenario, objetivo y personajes de la demostración final.

1.1 Creación, gestión, edición y uso de habilidades

Después de estudiar y analizar los diferentes juegos del género TRPG, decidimos incluir un sistema de creación de habilidades mediante el uso de editores de Unity, es aquí donde invertí la mayor parte del tiempo en el inicio del proyecto, ya que era necesario ofrecer total libertad al programador. Esto conllevo a desarrollar un editor sencillo y fácil de usar, pero que a su vez permitiese cualquier tipo de creaciones.

Además decidimos ofrecer la posibilidad de editar las habilidades existentes después de haberlas creado, por sí el resultado obtenido no era el deseado, por ello, añadí a este primer editor una segunda sección que permitiese modificar cualquier habilidad creada anteriormente.

La parte más complicada de las habilidades fue integrarlas con el juego, de tal manera que fuesen accesibles y comprensibles para el jugador, por ello decidimos implementar un menú de selección “ingame” que muestra todas las habilidades disponibles.

1.2 Creación del sistema de selección para IsoUnity

Cuando decidimos trabajar en la demostración gráfica, me encargué de desarrollar el sistema de selección de objetivos y celdas a la hora de realizar acciones, para ello hice uso del funcionamiento de IsoUnity, y mediante un algoritmo matemático, fui capaz de desarrollar un método que permitiese marcar las celdas disponibles para las diferentes acciones, de tal manera que el jugador entendiese cuales eran sus posibilidades en el momento de realizar cualquier acción.

Estoy muy orgulloso de este apartado ya que aquí es donde he podido unir muchos de los conceptos que he aprendido en todas las asignaturas de la carrera. Además el resultado conseguido es bastante comprensible.

1.3 Implementación del sistema de Turnos, acciones y movimientos.

Para que todo cobrase vida, centre mis investigaciones en los sistemas de turnos más utilizados en otros videojuegos y terminé con un resultado bastante interesante.

Con el objetivo de unir todo lo mencionado anteriormente, utilice un sistema que clasificaba a los personajes del juego según su atributo “Agilidad” y que gestionaba el funcionamiento de cada personaje, aunque el sistema está basado en uno anterior, añadí unos cambios que marcan la diferencia con otros sistemas de lucha. Decidimos que el jugador tendrá cuatro posibles acciones no repetibles y que el movimiento iría ligado a esta estructura de turno, evitando así el uso en exceso de habilidades o movimientos muy potentes.

1.4 Unión de los editores para su uso durante el combate.

Cuando todos los editores estuvieron listos, sobre todo los de Luis que eran mucho más complejos y debían ser más flexibles que los míos, mediante la implementación de unas clases, conseguí ligar todos los atributos e información de cada personaje a una entidad de IsoUnity, de tal manera que esa entidad representaba todo lo programado en los editores. Esta parte fue realmente

satisfactoria porque pudimos unir todos los “datos y números” de los editores en un objeto que respondía y actuaba según los parámetros programados.

1.5 Creación del escenario, objetivo y personajes de la demostración.

La parte final del proyecto me llevó a estudiar y buscar diferentes texturas y personajes para realizar el prototipo final que demostrase todo lo trabajado anteriormente, junto a Luis y los tutores, ideamos una temática entretenida y conocida que mostrase las diferentes opciones y utilidades que se han trabajado durante todo el curso.

2. Luis Alfonso González de la Calzada

2.1 Base de datos general.
2.2 Base de datos de personajes.
2.3 Conexión de la BD de personajes con el sistema de combate.

2.1 Base de datos general

Después de estudiar y analizar los diferentes juegos del género TRPG y las herramientas de edición (sobre todo RPGMaker) diseñé una base de datos única que centralizase y relacionase todos los elementos comunes del juego de forma que la creación de estos elementos fuese sencilla para el usuario y las dependencias se actualizasen en tiempo real.

Para ello generé un sistema de editores independientes que relacionasen los elementos creados (atributos, etiquetas, pasivas, clases y especializaciones, objetos ya configuración de slots) basados en un sistema de clases explicado en el Capítulo 3. Conseguir que los editores fuesen amigables, flexibles y completos fue una tarea ardúa ya que las dependencias entre los elementos eran delicadas y hubo que estudiar la documentación de Unity al respecto de la creación de editores.

Las clases explicadas en el Capítulo 3 requirieron de gran manejo de estructuras de datos (diccionarios, listas, ordenamientos…) y tuve que tener especial cuidado en no generar excepciones en ese sentido. Hice uso extensivo de la programación orientada a objetos con uso de plantillas y herencia además de tener que aprender a manejarme con C# (mi background es más de C++).

Esta base de datos general se ubica en memoria pero una vez se termina de trabajar con ella se puede guardar como un asset de Unity. Esta parte fue problemática ya que requirió que aprendiese a serializar elementos y consultar extensivamente documentación de Unity. Lo que parecía una carga pequeña llevó mucho más tiempo del previsto.

Esta parte es independiente del juego, lo cual me permitió desarrollar en paralelo con Javier mientras él se ocupaba del sistema de batalla y la base de datos de habilidades.

2.2 Base de datos de personajes

Afronté los mismos problemas que en la base de datos general pero con el extra de que el editor de personajes iba a usar toda la información que se generaba con los editores de la base de datos general (editores, clases, POO, serialización…).

Tuve que desarrollar un algoritmo de creación de ficha de personaje y adaptarlo al editor de personajes para que el proceso de crear un personaje fuese fácil pero a la vez potente para los futuros usuarios de la herramienta.

Esta parte también era independiente del juego, pero hubo que tener en cuenta que las fichas de personajes creados y que se guardan en la base de datos son el input para el sistema de batalla que es el que gestiona los personajes y sus atributos cuando están ingame.

2.3 Conexión de la BD de personajes con el sistema de combate

Esto se complementa con el punto 1.4 de Javier. Se diseño un gestor de fichas de personaje que ofrece a modo de API al sistema de batalla todo lo necesario para funcionar y poder actualizar los elementos en la base de datos de personaje con sus atributos y estadísticas actualizadas.

Referencias

[1] S. Miyamoto, “Super Mario Bros (Nes),” 1985. [Online]. Available: https://www.theguardian.com/technology/gamesblog/2.

[2] I. Epic Games, “Unreal Engine,” *Unrealengine.Com*, vol. 7, no. 11, 2015.

[3] J. G. Bond, *Introduction to Game Desing, Prototyping, and Development: From Concept to Playable Game - With Unity and C#*. 2014.

[4] V. M. P. Colado and I. P. C. Jose, “IsoUnity GitHub,” 2014. [Online]. Available: http://narratech.com/en/isounity/.

[5] U. Games, “Monument Valley Game,” 2014. [Online]. Available: https://www.monumentvalleygame.com/.

[6] “Chroma Squad Game,” 2015. [Online]. Available: http://www.chromasquad.com/.

[7] Y. Matsuno, “Final Fantasy Tactics - Game Boy Advance,” 2003. .

[8] “Fire Emblem Series,” 1990. [Online]. Available: http://fireemblem.wikia.com/wiki/Fire_Emblem_(series).

[9] Y. Matsuno, “Final Fantasy Tactics Franchise.” [Online]. Available: https://www.grouvee.com/games/franchise/1351-final-fantasy-tactics/.

[10] Roy, “SquareSoft to Square Enix.” [Online]. Available: http://www.otakufreaks.com/de-squaresoft-a-square-enix-i-historia-de-una-fusion/.

[11] A. Kawazu, “Ivalice Alliance.” [Online]. Available: http://finalfantasy.wikia.com/wiki/Ivalice_Alliance.

[12] A. Blizzard, “Activision Blizzard Information,” 2008.

[13] I. Barnard, “The Elder Scrolls V: Skyrim - #7 RPG,” *MELUS*, 2012. [Online]. Available: http://www.ign.com/top/rpgs/7.

[14] G. G. Games, “Path of Exile Game,” 2013. [Online]. Available: https://www.pathofexile.com/.

[15] Supergiant Games, “Bastion Game,” 2011. [Online]. Available: https://www.supergiantgames.com/games/bastion/.

[16] A. Rao and G. Simon, “Supergiant Games,” 2009. [Online]. Available: https://www.supergiantgames.com/.

[17] P. Yang, B. Harrison, and D. L. Roberts, “Identifying Patterns in Combat that are Predictive of Success in MOBA Games,” *Proc. Found. Digit. Games 2014*, pp. 1–8, 2014.

[18] J. A. Diaz, “The Walking Dead,” *MELUS*, vol. 32, no. 3, pp. 261–263, 2007.

[19] Scopely, “The Walking Dead Road to Survival,” 2015. [Online]. Available: http://scopely.com/game/the-walking-dead/.

[20] M. Schulzke, “Moral decision making in fallout,” *Game Stud.*, vol. 9, no. 2, 2009.

[21] G. Gygax and D. Arneson, “Dungeons and Dragons,” *J. Read.*, no. 1991, pp. 1–24, 2001.

[22] B. James Ohlen, Ray Muzyka, “Baldur’s Gate Video game.” [Online]. Available: https://www.baldursgate.com/.

[23] Hidalga Erenas, “Alineamiento de personalidades,” 2015. [Online]. Available: https://realidadesalternarrativas.wordpress.com/2015/03/03/alineamiento-personajes/.

[24] S. Camarotti, “Behold Studios,” 2009. [Online]. Available: http://beholdstudios.com.br/.
[25] B. Berthier and L. Parisot, “Video games overview,” *Psychotr.*, vol. 18, no. 3–4, pp. 25–43, 2013.
[26] S. Powered, “Steam Greenlight - Direct,” 2016. [Online]. Available: https://partner.steamgames.com/steamdirect.
[27] K. T. Kazuhiru Nishi, “Ascii Corporation.” 1977.
[28] B. Vogler, “Wurfel Engine,” 2015. [Online]. Available: http://wurfelengine.net/.
[29] I. Hamilton, “JSIso,” 2014. [Online]. Available: http://jsiso.com/.
[30] J. (Github), “IsoHill,” 2012.